A FORMAL DEFINITION OF THE OBJECT-ORIENTED PARADIGM FOR REQUIREMENTS ANALYSIS

THESIS

Andrew D. Boyd, Captain, USAF

AFIT/GSS/ENG/91D-3

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GSS/ENG/91D-3

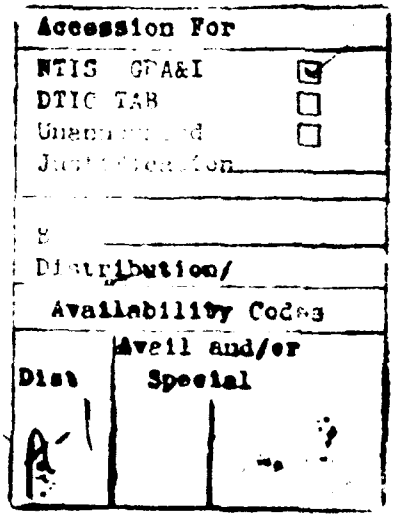
A FORMAL DEFINITION OF THE OBJECT-ORIENTED PARADIGM FOR REQUIREMENTS ANALYSIS

THESIS

Andrew D. Boyd, Captain, USAF

AFIT/GSS/ENG/91D-3

The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.



AFIT/GSS/ENG/91D-3

A FORMAL DEFINITION OF THE OBJECT-ORIENTED PARADIGM FOR REQUIREMENTS ANALYSIS

THESIS

Presented to the Faculty of the School of Systems and Logistics

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Software Systems Management

Andrew D. Boyd,

Captain, USAF

December 1991

Acknowledgments

I would like to take this opportunity to thank Dr. David Umphress for being the inspiration behind this research to develop a formal definition of the Object-Oriented paradigm, and for extending to me an invitation to use it for my thesis. Working with him has served to enrich my understanding of both Object-Oriented computing and the benefits of formalisms; two areas that are gaining much popularity in this time of the "software crisis." Without the knowledge and guidance of Dr. Umphress, this research would not have been possible. Much of the credit for this thesis goes to him. I am also deeply indebted to Dr. Henry Potoczny for providing guidance to us in the area of discrete math and formal definitions. I would also like to thank Capt Dedolph for providing a different point of view on the thesis.

I will also take this opportunity to express my gratitude to my wonderful wife Nancy, and marvelous son Alex. Without the help and support they both gave me, this thesis would have been even more difficult than it already was. I would also like to thank the one who is not here yet for staying not here yet so that I could finish this thesis before you got here.

To all of you, thank you.

Andrew D. Boyd

Table of Contents

Page

Page

List of Figures

Page

List of Tables

Page

Abstract

This paper develops a formal definition of the Object-Oriented paradigm for requirements analysis. The literature was surveyed for both formal and informal methods for conducting an Object-Oriented Requirements Analysis (OORA). The informal methods reviewed are: Bailin's, Shlaer and Mellor's, Booch's, and Coad and Yourdon's. The formal methods reviewed are: Bralick's, Z, and REFINE. None of the methods were found to be adequate for doing an OORA. A formal definition of an OORA, based on the concept of classes, is developed. The definition itself is presented as set and relation theory. A supporting graphical representation is also developed and presented. The graphical method allows a system to be successfully leveled. The formalism is validated by applying it to the Air Traffic Control (ATC) simulation.

A FORMAL DEFINITION OF THE OBJECT-ORIENTED PARADIGM FOR REQUIREMENTS ANALYSIS

I. Introduction

It has been said that as much as 10% of the entire Department of Defense (DOD) budget is spent on the development and maintenance of weapon system software, with 80% of that number going towards the labor intensive tasks of reworking and updating the software (Goldberg, 1990:60). Rework is usually performed to remove errors in the software, either in the code or in the design, and updates are adding new requirements or previously missed requirements. According to (Bailor, 1991), the source of these software errors are:

1) Requirements and specification: 11 percent
2) design or design related: 81 percent
3) language and environment: 6 percent
4) other: 2 percent.

Two items of interest have recently emerged that could have a significant positive impact on this situation. They are the Object-Oriented (O-O) paradigm and the use of formal methods in software development. Between the use of these two concepts, a significant number of errors could be avoided and thus reduce the amount of rework and updates required.

As the O-O paradigm rapidly gains popularity as a programming methodology (Yelland, 1989:290) the need for conducting Object-Oriented Requirements Analysis (OORA)

grows as well. A requirements specification produced by an OORA is the ideal front end to an Object-Oriented Design (OOD) (Booch, 1991:37), and the products of an OOD are then used to completely implement an Object-Oriented Program (OOP) (Booch, 1991:141). Thus, to produce the best OOP, one should begin with a detailed analysis (OORA) to produce the best specification. The most effective technique for assisting in detailed analyses is the formal technique (Sommerville, 1989:125). This thesis combines the O-O paradigm and the formal technique to develop a formal definition of the O-O paradigm for requirements analysis.

Background of the Object-Oriented Paradigm

The term "object-oriented" (O-O) connotes the process of modeling a system as "a set of autonomous agents that collaborate to perform some higher level behavior" (Booch, 1991:15). The idea of interacting agents is the fundamental difference between the object-oriented view and the more traditional structured-oriented view of the software development process in which the system under consideration is modeled as a collection of functions. These interacting agents are known as "objects" (Booch, 1991:15).

Object-Oriented Programming. The fundamental ideas of O-O, classes and objects, appeared first in the programming language Simula 67 and were later refined in languages such as the Flex system and Smalltalk (Booch, 1991:34). Recently, a large number of objected-oriented programming languages have been developed. These include C++ and Objective C, Ada, Object Pascal, Eiffel, and many dialects of Lisp (Booch, 1991:35).

Object-Oriented Design. As OOP became increasingly popular, techniques for designing software to fit the programming paradigm had to be developed. These techniques came to be known as OOD. Some techniques were very language oriented (e.g. Booch)

(Booch, 1987), and some, such as Jackson Structured Development, were a mix of structured and object-oriented methodologies (Henderson-Sellers and Edwards, 1990:146).

Object-Oriented Requirements Analysis. Prior to OORA, requirements analysis was often done using a structured (functional) decomposition and therefore, in a form inappropriate to OOD (Umphress and March, 1991:1). Now, the informal techniques for doing OORA are as varied as those for doing OOD and OOP and range from a mix of structured and object-oriented views (Bailin, 1989), to relational database theory (Shlaer and Mellor, 1988; Shlaer and Mellor, 1989), to pure object-oriented (Coad and Yourdon, 1990). These informal methods do not agree with each other and they do not provide a sound basis upon which to build an adequate requirements specification. A formal based method for doing OORA would provide a mathematically sound basis upon which to build a requirements specification. However, no formal based methods currently exits for conducting an OORA.

Benefits of Formal Methods

Formal methods provide the software analyst with the ability to precisely define software in terms of "what" the software must do rather than "how" the software must do it (Spivey, 1989:1). Sommerville provides the following six benefits of formal specifications (Sommerville, 1989:126-127):

1) Formal specifications provide insights into and understanding of the software requirements.

2) Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specification.

3) Formal specifications may be automatically processed.

4) It may be possible to animate a formal system specification to provide a prototype system.

5) Formal software specifications are mathematical entities and may be studied and analyzed using mathematical methods.

6) Formal specifications may be used as a guide to the tester of a component in identifying appropriate test cases.

Research Objectives

The overall objective of this thesis is to develop a formal definition of an OORA. However, that objective will be accomplished through the following sub-objectives:

1) Determine the necessary concepts of the O-O paradigm that must be incorporated into a formal definition for an OORA.

2) Develop a graphical representation of the key concepts to supplement the formal definition.

3) Determine the mathematical representation of the individual key concepts to be used in the formalism.

4) Produce the formal definition by defining the mathematical relationships between key concepts.

5) Validate the research through a sample problem.

Scope

The formalism developed in this thesis is applicable only to the requirements analysis phase of software being developed in the context of the object-oriented paradigm. The formal definition and supporting graphical representation can be applied only after an analysis has been conducted to initially determine the system classes, structure, and dynamics. This formalism provides the capability to produce a requirements specification capable of being used as inputs to the design phase of the software development process.

Overview

The remainder of this thesis is broken down into four chapters. Chapter II is a survey of the literature on current schools of thought in the field of OORA. Chapter III develops the formalism with illustrated examples from the Spelling Checker System (see Appendix). Chapter IV is the application of the formalism to the Air Traffic Control (ATC) simulation. Chapter V presents a summary of the research, conclusions reached from the research, and recommendations for future research.

II. Literature Survey

This chapter is a survey of the literature on OORA. The major schools of thought in the area of OORA for both informal and formal definitions are presented.

Major Schools of Thought on OORA

Since OORA is a relatively new field, there are few truly original perspectives on the subject. This section presents those in the literature for both informal and formal methods of accomplishing OORA.

Informal. According to (Umphress 1991) "an 'informal method' connotes a seat-of-the-pants, ad hoc approach to analysis; i.e, few rigid principles are used." In the case of OORA, informal approaches range from a hybrid of structured and object-oriented methodologies (Bailin) to everything being described in terms of an object (Coad and Yourdon). The following discussion begins with the hybrid approach and works towards the pure object approach.

Bailin. Bailin's Object-oriented Requirements Specification (OORS) (Bailin, 1989) is a hybrid of structural and object-oriented methodologies. He describes the key abstractions of OORA as entities and functions. The contents of an entity is described by data structures, the underlying state of some process as it evolves in time, and the aspect of the process that is persistent across repeated execution cycles (Bailin, 1989:609). Functions, on the other hand, transform inputs to outputs but have no underlying state that persists after the function is complete. Functions occur within entities and are performed by or act on the entity.

Bailin (1989:609) describes a seven step approach for OORS:

1) Identify key problem-domain entities.

2) Distinguish between active and passive entities.

3) Establish data flow between active entities.

4) Decompose entities (or functions) into sub-entities and/or functions.

5) Check for new entities.

6) Group functions under new entities.

7) Assign new entities to appropriate domains.

The first three steps are only performed once; the remaining steps are performed iterativly until the desired level of detail is attained.

The initial problem domain entities are determined by drawing structured analysis Data Flow Diagrams (DFDs) and then extracting the nouns from the process names (step 1) (Bailin, 1989:610). Entities can be classified as active or passive. Bailin distinguishes between active and passive entities (step 2) as: "An active entity is an entity whose functions we want to consider in the analysis phase. A passive entity is one whose functions we do *not* want to consider until the design phase" (Bailin, 1989:612). Next, Entity Data Flow Diagrams (EDFDs) are constructed (step 3) by specifying active entities as processes and passive entities as data flows or data stores. The highest level EDFD contain only entities whereas lower level EDFDs contain both entities and functions. Entities are further decomposed into subentities and/or functions and functions decomposed into subfunctions in step 4.

Throughout the process of iterative decomposition, Bailin's method checks to ensure every functional requirement can be met by one or more of the functions identified in the EDFDs. If additional functions are required, new entities may need to be added and the

functions grouped with it (steps 5 and 6). The final step (step 7) maintains a source of domain analysis information for use by other projects.

Bailin approaches the O-O philosophy by adding services and attributes to his functional foundation. He points out that services may be a group of related functions that are identified with an entity and represent some sort of user access point to that entity. With respect to attributes, Bailin writes,

> In the terminology of an E-R model, state variables are attributes of an entity. Since an entity (object) provides functions (methods) as well as state variables, it seems reasonable to view these functions as attributes of the entity. . . . Every functional attribute of an entity would have to appear as a function in the EDFD that decomposes that entity. Every Data-valued attribute of an entity would have to appear as a data store in the EDFD. (Bailin, 1989:618-619)

This approach to OORA relies heavily on the structured technique of Data Flow Diagrams while combining the idea of entities that contain functions and data interacting with each other. The resulting EDFDs contain functions, entities, and data.

Shlaer and Mellor. Shlaer and Mellor's approach to Object-Oriented Analysis (OOA) is based on the development of an information model derived from database relational theory (Shlaer and Mellor, 1988:xii, 13). State models and process models complete the analysis by representing the behavior of the information model (Shlaer and Mellor, 1989:66).

The first step in their OOA is the construction of a model to represent the system under development. The model, called the information model, illustrates the main components of the system and how they relate. The first part of the modeling process is to determine the objects in that system. An object is defined as "an abstraction of a set of real-world things such that all of the real-world things in the set -- the instances -- have the same characteristics; and all instances are subject to and conform to the same rules" (Shlaer and Mellor, 1989:67). Objects can be tangible things, roles, specifications or quality criteria,

aggregations of equipment, or even steps in a process found in the problem domain. Once objects are identified, attributes are added to describe a single characteristic possessed by all the instances of that object. For example, a Vehicle Identification Number might be an attribute of a vehicle object since all vehicles have one. Finally, object-to-object multiplicity and interdependency relationships are added to complete the model. Shlaer and Mellor define a relationship as "an abstraction of a systematic pattern of association that holds between real-world things" (Shlaer and Mellor, 1989:67). An attribute in one object that is referenced by another object is called a referential attribute. Extra objects are added to capture the referential attributes that are a result of many-to-many relationships between objects (Shlaer and Mellor, 1989:69-70).

The next major step is to develop state models. State models represent the dynamic behavior of the objects and relationships. Shlaer and Mellor write that each object follows a lifecycle. Some lifecycles may be trivial (e.g. the object exists) and some may be very complex (e.g. the object goes through several stages in it's lifecycle) (Shlaer and Mellor, 1989:70). A lifecycle may be made up of a number of stages that must obey specified physical laws, operating policies, and rules. Each stage in the lifecycle represents a state of that object. An object moves from one stage to the next when the object is triggered by an event. Events may be timed (e.g. after a process has run for a timed interval, the object will proceed to the next stage) or may be requests from other objects (e.g. another object requesting a change in it's state, thus progressing it through it's stages) (Shlaer and Mellor, 1989:72). Actions occur within the object in response to an external (to that object) event. After the state models are developed, coordination of the lifecycles must be accomplished to complete the dynamic description of the system. (Shlaer and Mellor, 1989:70-73)

The final step in Shlaer and Mellor's OOA is the development of a set of process models. The process models use data flow diagrams to define information processing of each state in the state model. Little original information is added to the analysis by this step but it is intended to allow the analyst to see areas of potential process reuse. (Shlaer and Mellor, 1989:75-76)

The key concepts of the Shlaer and Mellor approach are that object-oriented systems can be described as a relational database model that consist of objects, attributes describe the objects, and relationships describe object-to-object dependencies. The addition of state and process models serve to represent the dynamic behavior of the system.

Booch. Grady Booch writes, "Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain" (Booch, 1991:37). Although Booch's contribution to the O-O paradigm has been in the designing and programming aspect of the software lifecycle, his initial steps for designing object-oriented systems can be viewed as object-oriented analysis since he claims "the boundaries between analysis and design are somewhat fuzzy" (Booch, 1991:141). Booch (1991:190) uses four steps in developing object-oriented systems:

1) Identify the classes and objects at a given level of abstraction.

2) Identify the semantics of these classes and objects.

3) Identify the relationships among these classes and objects.

4) Implement these classes and objects.

The first step involves the identification of candidate classes from the vocabulary of the problem domain (key abstractions) and the determination of "important mechanisms that provide the behavior required of objects that work together to achieve some function" (Booch, 1991:191). The second step defines the behaviors of each class identified in the previous

step. The key to this step is viewing each class from the perspective of its interface to determine what can be done to each instance of that class (Booch 1991:192). The accomplishment of a behavior leads to a change in the state of a class and is thus best represented by a state transition diagram and timing diagrams (Booch 1991:167-168, 173-174).

The third step identifies the specific types of relationships that exist among classes (Booch, 1991:193). Booch refers to three basic kinds of class relationships. These relationships include a-kind-of (AKO), where one class is a specialization of another class (e.g. a Car is a-kind-of Vehicle); aggregation, where one class is a part of another class (e.g. an engine is part of a car); and association, where classes are associated by some semantic connection among otherwise unrelated classes (e.g. a person controls the car) (Booch, 1991:96). The simplest way to describe an association is the relationship between two classes that can communicate with each other but do not in any way contribute to the state of each other. Objects interact by passing messages between each other. The messages serve to trigger behaviors within the object receiving the message (Booch, 1991:169-172).

The fourth step takes a more detailed look inside the classes identified previously as the system is decomposed to the lowest level necessary for adequate requirements analysis (Booch, 1991:195).

These four steps are an incremental and iterative process that continues until a level of detail has been identified that fully defines the requirements of the system being modeled (Booch, 1991:190).

After all of the classes have been identified, it is possible to move one level of abstraction above the class to what Booch calls class categories. Class categories are a means of organizing classes into meaningful chunks, or more appropriately, subsystems. Class

categories are the highest level of abstraction for the system and allow the developer to understand the general logic of the system.

> Each class category denotes another class diagram; thus, if we pick any one of the class categories from this topmost diagram and zoom into its implementation, we may find either a simple class diagram consisting of classes, class relationships. . . or (for very large problems) another class diagram containing other class categories, which in turn represent other object diagrams. (Booch, 1991:162)

Class categories serve to further encapsulate information from the rest of the system by only providing visibility to internal classes that can be used by outside clients (Booch, 1991:162).

Booch's main contribution to the O-O paradigm is in design. However, his front end to design makes the assumption that an OORA has not been accomplished, and therefore, classes, their relationships, behaviors, and states must be determined before the OOD process can begin. When Booch describes his method to fill in the missing information to begin an OOD, he is actually describing an OORA. Thus, Booch's OOD has an OORA built in.

Coad and Yourdon. The most comprehensive work to be found on object-oriented requirements analysis had been done by Peter Coad and Edward Yourdon. Coad and Yourdon use a five step approach to complete OORA. The five steps are identifying objects, identifying structures, defining subjects, defining attributes, and defining services (Coad and Yourdon, 1990:34). Their model is presented in five layers and are called the subject, object, structure, attribute, and service layers (Coad and Yourdon, 1990:35).

The subject layer is an abstraction mechanism that controls how much the reader considers at one time. It can be seen as describing the subsystem level of the system being modeled. Subjects can be combined at higher levels to suppress unnecessary detail from the reader (Coad and Yourdon, 1990:188).

The object layer is the level below the subject layer. It contains the objects that represent the problem space. Objects are "an abstraction of data and exclusive processing on that data, reflecting the capabilities of a system to keep information about or interact with something in the real world" (Coad and Yourdon, 1990:177).

The structure layer connects objects together in two different ways. The classification structure relates those objects that are specializations of a general object while the assembly structure relates the composite, or "has-parts", structure of objects (Coad and Yourdon, 1990:183).

The attribute layer consists of defining data elements within objects that define the object's state space. Additionally, this layer serves to connect objects together that require the services of other objects as well as showing the multiplicity and participation (optional or mandatory) of all of the objects (Coad and Yourdon, 1990:190-191).

The final layer, the service layer, describes the behaviors each object can exhibit and is also where intra-object communication is shown. Behaviors are described as services listed within the objects; communication is represented by directed dashed lines (arrows) between objects (Coad and Yourdon, 1990:127-138). Coad and Yourdon use state-event-response tables to enhance the dynamics of this layer (Coad and Yourdon, 1990:134-135).

This approach is the most truly object-oriented approach found in the literature in the sense that it requires objects, attributes of objects, services to be provided by the objects, the structure of the relationships between objects, and the higher level grouping of objects called the subject layer.

Formal. A formal method is a method that when applied to the accomplishment of a particular task, can be proven to be correct. Formal methods also include "a well-defined, controlled vocabulary" (Umphress, 1991). Formal methods, much like the informal methods,

range from loose methods like (Bralick, 1988) to executable specifications like REFINE. The Z (pronounced "zed") specification language does not produce executable specifications, but it is considered to be one of the most widely used specification languages. The following discussion on formal methods begins with the method presented by Bralick, then discusses Z, and finally REFINE.

Bralick. Bralick defines an object as having "a unique identity, composed of a set of attributes, a set of behaviors, and a set of (sub)objects" (Bralick, 1988:3.5). An attribute, in Bralick's definition, represents some value that is a property of an object that serves to describe or supplement the meaning of the object (Bralick, 1988:3.6). A behavior is an activity or operation of objects whose result is the modification of some set of attributes of that object (Bralick, 1988:3.8). Objects communicate with each other by sending messages requesting their attributes be adjusted by means of an object's behavior (Bralick, 1988:3.20).

The key to Bralick's object model is that he considers objects as consisting of a unique identifier, attributes, behaviors, and other objects. Although Bralick labeled his object model as a formal model, it lacks the rigor of a true formal method.

Z. Z is known as a model-based formal specification language developed by J. R. Abrial for developing formal specifications of software systems requirements (Sommerville, 1989:158). "Model-based specification is a technique that relies on formulating a model of the system using well understood mathematical entities such as sets and functions. System operations are specified by defining how they affect the overall system model" (Sommerville, 1989:158). Z is based on typed set theory since sets are mathematical entities that are formally defined and well understood (Sommerville, 1989:158).

The basic building block of Z is the schema. A schema has three parts, a name; a signature, which introduces some specification entities; and a predicate part, which defines

relationships between the entities within the signature (Sommerville, 1989:159). An Entity in the signature is described by the set of values it can assume (e.g. set of natural numbers). "Every Z specification begins with certain objects [entities] which play a part in the specification, but have no internal structure of interest. These atomic objects [entities] are the members of the *basic types* or *given sets* of the specification" (Spivey, 1989:27). Predicates (relationships) are defined over the entities in the signature which must always hold (Sommerville, 1989:159). Schemas can be combined to form more complex schemas where the new schema inherits all of the signature and predicate parts of the its constituent schemas (Sommerville, 1989:160).

Operations can be described using schemas by annotating input and output entities within the signature and describing, in the predicate part, the state change relationship (i.e. precondition and postcondition) between entities within the signature (Sommerville, 1989:161).

The domain and range of functions are used in Z to describe the set of valid inputs and the set of associated outputs to a given relationship. Z provides a number of operators which allow the domain of the functions to be manipulated (Sommerville, 1989:164-165).

The completed specification produced by Z is a set of schemas that will describe mathematically the system being modeled.

REFINE. In contrast to Z which to date can only be accomplished by hand, REFINE is considered to be a specification language that will produce executable specifications (Reasoning Systems, 1990:1-2). REFINE provides specification programming to include set theory, logic, transformation rules, pattern matching, and procedure (Reasoning Systems, 1990:1-2).

A specification is developed as a collection of top-level definitions using the REFINE language. The language itself is in the form of set and logic notation which has the capability of defining "a type, variable, constant, action, assertion, explicitly-defined or assertionally-defined function, rule, or gammer" (Reasoning Systems, 1990:7).

Specification descriptions can be modeled using REFINE and then stored in an object base (object-oriented database) for later reuse (Reasoning Systems, 1990:1-3). Specifications are continually refined (more explicitly defined) using the tools of the language until they actually become the executable code when compiled.

REFINE claims to support the O-O paradigm by defining object classes:

> An object-oriented model of an application domain specifies the classes of entities in the domain, and the attributes that are applicable to the entities in each class. An entity of an application domain may be represented in REFINE by an object, i.e. an element of the type **object**. An entity class is represented by a variable of type **object-class**, which is a predicate on objects. An attribute that applies to any element of an entity class is represented by a variable of type **map** whose domain is the object class representing the entity class. (Reasoning Systems, 1990:10)

The REFINE interpretation of an object-oriented system consists of objects, object-classes, and attributes. An O-O specification would consist of top-level definitions of those entities. These definitions would not only define the entities themselves, but would also define the dynamics of the system.

Summary of the Object-Oriented Models

The Softwar Engineering Institute's (SEI) module "Software Specifications: A Framework" describes four aspects of a specification that should be addressed when analyzing software. They are behavior, interface, flow, and structure (Rombach, 1989:9-11). Behavior is defined as "the externally observable response of a product or process to stimuli in actual

use" (Rombach, 1989:10); interface is defined as "the structure of the boundary between product or process and its environment" (Rombach, 1989:10); flow is defined as "the internal dynamics of a product or process in actual use" (Rombach, 1989:10); and structure is defined as "the organization of a product or process into interacting parts" (Rombach, 1989:11).

Table 2-1 puts the object-oriented methods presented in perspective with the SEI framework.

Table 2-1. Comparison of Object-Oriented Models

Specification Aspects

SEI Aspect	Object-Oriented Models				
	Bailin	Shlaer & Mellor	Booch	Coad & Yourdon	Bralick
Interface	Data flows	State Model	Class Template	Service Layer	--
Behavior	Active Entity Function	State Model	State Transition Diagrams Timing Diagrams	State-Event-Response	Operation
Flow	Entity Data Flow Diagram (EDFD)	Process Model	Message	Communication Arrows	Message
Structure	Sub Entity Sub Function	Referential Attributes Relations	A-Kind-Of Aggregate Association	Has_part A-Kind-Of Instance Relations Attributes	Attributes Sub-objects

III. Model Development

Chapter III describes the development of the formal model of an object-oriented requirements analysis. The first section of this chapter presents an operational definition of the model and defines the graphical representation. The second section presents the formal definition itself.

Operational Definition

From an analysis perspective, an object-oriented system may be conceptualized as:

> An object-oriented system is composed of classes. Each class has a unique name, one or more parent classes, an interface part and a hidden part. The interface part describes those aspects of a class that need to be known by a client in order to use this class. The hidden part describes the detailed view of the class to include the relationships and restrictions on those relationships with other classes.

Key to this definition are the terms class, name, parent, interface part, and hidden part. Each term is explained below. The Spelling Checker System (SCS) (see Appendix for the complete SCS analysis) is used to illustrate the major principals.

Class. The class is the basic building block of the O-O analysis. It is a generalization (abstraction) of a group of similar concepts. Classes do not have state, but describe the possible range of states for the system. Two icons represent classes: one shows that the interface is being viewed, and the other shows that the hidden part is being viewed. Figure 3-1 is the class interface icon and Figure 3-2 is the class hidden icon. The most noticeable difference between the two icons is the "shadow-box." The icon with the shadow represents the interface of the class whereas the icon without he shadow represents the detailed view of the class that is hidden behind the interface. The concept of

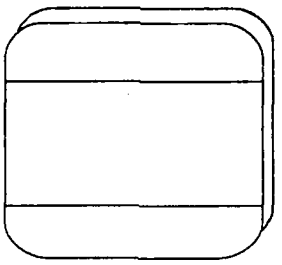


Figure 3-1. Class
Interface Icon

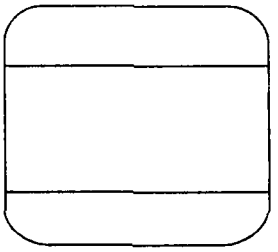


Figure 3-2. Class
Hidden Icon

going from the abstract (the interface view) to the detailed (the hidden view) is referred to as leveling (Davis, 1990:59-60; Coad and Yourdon, 1990:80).

Object. An instance of a class that contains a particular state is called an object. An instance of a class that contains a state range is just another class. If a class is said to contain a range of states (a set), and an object is the realization of a specific state within that range, then a class can be said to contain a set of objects that fall within that range of state. Objects need only be considered in the most abstract sense during analysis but become more prominent during the successive stages of software development.

Name. The top part of the class icon is called the "name area." It names the class represented by the icon and is the same for both interface and hidden icons. The name of a class represents the unique identifier of that class and is the way to distinguish that class from other classes. It also allows the analyst to provide a meaningful description of what the class represents.

Parent. The parent concept denotes an inheritance relationship. All classes are derived from at least one parent class. The parent of a class passes all of its interface and hidden aspects to the class being defined. The class, in turn, can add information to make it a specialization of its parent. All classes have, as a minimum, the Universal class as an ancestor; although, not necessarily as a direct parent. The Universal class is the highest level of abstraction and encompasses all classes. The Universal class, represented as *U*, has no parent and its interface and hidden parts are empty. The relationship between a parent and its child is an "a-kind-of" relationship and will be discussed in more detail later.

Interface Part. The interface to a class describes that information which a client would need to know in order to use this class. This interface can be divided into two parts, the interface class description and the capabilities.

Interface Class. Interface classes, listed in the middle area of the interface icon, denote those classes that must also be present in order for a client to use this class. One way to view an interface class is to liken it to a "wire" attached to the class. In order to use that class, all of its wires must be attached, and they must be attached to the classes listed in its interface. For example, a radio speaker would list a radio in its interface and would therefore need its "wires" attached to a radio in order for it to be of any use.

A class's interface class description determines what kind of class it is. Classes come in three levels of complexity: canonical, simple, and complex. Classes that have a higher

level of complexity are less easily reused than classes at lower levels of complexity. In addition, complex classes are more cumbersome to use then those with less complexity. The higher the complexity, the more information (in the form of other classes) needs to be present for a client to be able to use the complex class. In other words, in order for a client to use a complex class, every interface class of the complex class must be accessible by the client.

Canonical Class. Canonical classes are the most basic of all classes and have no interface classes. Canonical classes exist at the boundary between classes and values. The description of a canonical class represents the possible range of states that an instance of the class could take on. This capability allows the modeler to solve the problem of whether or not values are considered to be objects (MacLennan, 1982). MacLennan's argument is that values "amount to timeless abstractions for which the concepts of updating, sharing and instantiation have no meaning" (MacLennan, 1982:9) while objects can be instantiated, changed, have state, created and destroyed, and shared (MacLennan, 1982:11). The actual value would be associated with one or more instances of that class (objects) and would represent the state of these objects. There is only one "value" (i.e. the number 1) but it may be associated with several different objects that have that value as their state at any given time. For example, in the SCS, a canonical class *String* is defined to be the set of all combinations of the ASCII character set. An instance of a class associated with String would map to a specific member of that set. There may be several instances of the string "my" all mapping to the single value "my." The name of a canonical class as written in the "name-area" of the interface icon has only the first letter capitalized. A canonical class will only have an interface part since it represents the lowest level of abstraction that is necessary for the analysis, and thus no need to view the details of any hidden part.

Simple Class. A simple class is one that has only canonical classes in its interface. For example, the *TOKEN* class in the SCS is a simple class which only has the two canonical classes *String* and *Boolean* in its interface. Figure 3-3 is the interface representation of the simple class *TOKEN*. By convention, the names of simple classes are written entirely in upper case letters.

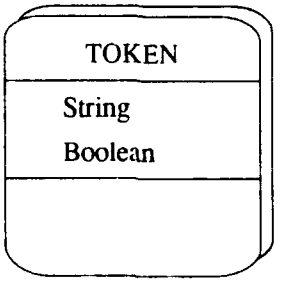


Figure 3-3. A Simple Class Interface

Complex Class. A complex class has at least one non-canonical class within its interface, and represents the most complex form of the class to use. The more non-canonical classes there are in a class's interface, the higher the module complexity and coupling and the less reusable that class will tend to be. The *SPELLING_CHECKER* class is an example of a complex class. It has three non-canonical classes as well as one canonical class within its interface. They are *USER*, *INPUT_DOC*, *OUTPUT_DOC*, and *String*. Figure 3-4 is the interface representation of the complex class *SPELLING_CHECKER*.

Capabilities. Capabilities, listed in the lower area of the interface icon, are those operations that a client can request this class to perform. The only capabilities that canonical classes posses are: *value_of*, *set_value*, *image_of*, and *convert_image*. Figure 3-5

shows the interface icon for the canonical class *Line_number* and Table 3-1 describes what those capabilities are.

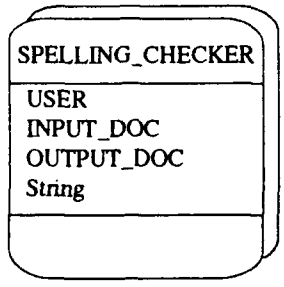


Figure 3-4. A Complex Class Interface

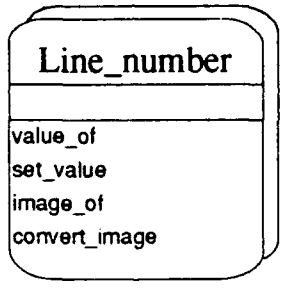


Figure 3-5. Canonical Class Capabilities

Hidden Part. The hidden part of the class describes a more detailed view of the class. From this view, all the classes that are required to be present in order for this class to accomplish its intended purpose are shown. The middle area of the hidden icon lists only those canonical classes that contribute state to the class. The lower area of the hidden icon lists all of the behaviors of this class including those listed as capabilities in the interface.

Additionally, the specific relationships that exist between this class and others are shown along with their multiplicity and participation restrictions.

Table 3-1. Canonical Class Capabilities Description

Capability	Description
value_of	Returns the value of the object in its internal representation (e.g. 11111111)
set_value	Sets the value of the object by setting its internal representation (e.g. My_Number := 11111111)
image_of	Returns a universal string representing the character value of the object (e.g. 11111111 would return 256)
convert_image	Takes a character value and returns an internal representation (e.g. takes 256 and return 11111111)

Relationships. There are five different class to class relationships. They are *a-kind-of*, *has_part*, *has_attribute*, *has_association*, and *has_domain*.

A-kind-of. The *a-kind-of* relationship (Booch, 1991:54) or classification structure (Coad and Yourdon, 1990:79-82) denotes an inheritance relationship between two or more classes. This relationship can also be thought of as a parent-child relationship where the child class inherits all of the parent's interface and hidden parts. In addition to having

its own unique name, the child class may add additional classes and capabilities to its interface to further make it a specialization its parent class. Child classes do not contribute in any way to the state range of its parent class. However, since this is an inheritance relationship, the child's state range is affected by that of its parent in addition to any state range that had been changed or added as a result of the specialization. Figure 3-6 shows the graphical representation of the *a-kind-of* relationship between the two classes *INPUT_DOC* and *OUTPUT_DOC* and their parent class *DOCUMENT*.

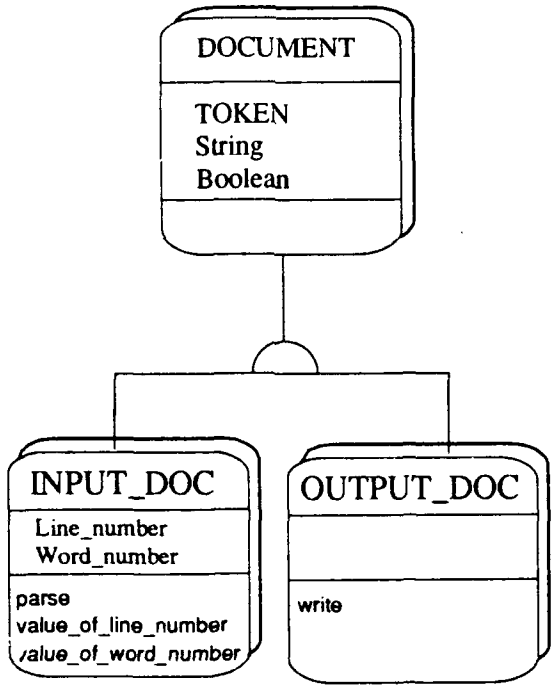


Figure 3-6. A-Kind-Of Relationship Notation

It can be seen from Figure 3-6 that the interface classes for *INPUT_DOC* are not only *Line_number* and *Word_number*, but also the classes *TOKEN*, *String*, and *Boolean* from its parent. The interface classes for *OUTPUT_DOC* are only the classes *TOKEN*, *String*, and

Boolean from its parent. Similarly, the capabilities of *INPUT_DOC* and *OUTPUT_DOC* are only those listed within their respective capability areas since their parent possessed none of its own.

Has_part. The *has_part* relationship denotes a structural relationship between two classes. Booch terms this type of relationship an aggregate (Booch, 1991:58-59) whereas Coad and Yourdon call this an assembly structure (Coad and Yourdon, 1990:82-92). The class that is the constituent ("part") contributes its state range to that of the class that is the aggregate. An example of this type of relationship is the *DOCUMENT* class. A *DOCUMENT* class has one part: a *TOKEN* class. In other words, a *DOCUMENT* is made up of *TOKEN*s. Figure 3-7 shows this relationship and its graphical notation.

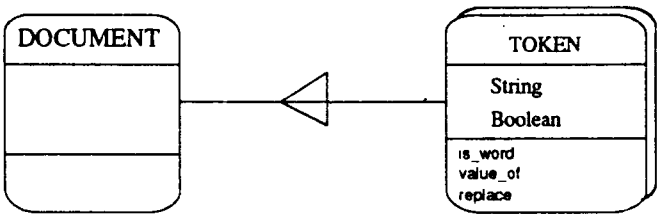


Figure 3-7. Has_part Relationship Notation

The constituent class, *TOKEN*, inherits nothing from the aggregate class, *DOCUMENT*. The *has_part* relationship is intended to show structural decomposition of the system being modeled.

Has_attribute. An attribute describes characteristics of a class that all instances of that class will inherit (Shlaer and Mellor, 1989:67). In other words, a

has_attribute relationship between two classes denotes a relationship where the presence of one class serves to describe some characteristic of another class. An attribute can be either a simple class or a complex class but not a canonical class since canonical classes only represent a mapping to values and are not descriptive in nature. An attribute contributes its state range to that of the class being described. An example of the *has_attribute* relationship is shown in Figure 3-8 where the *SPELLING_CHECKER* class has the *TOKEN* class as an attribute. The *TOKEN* class describes something about the state of the *SPELLING_CHECKER* class.

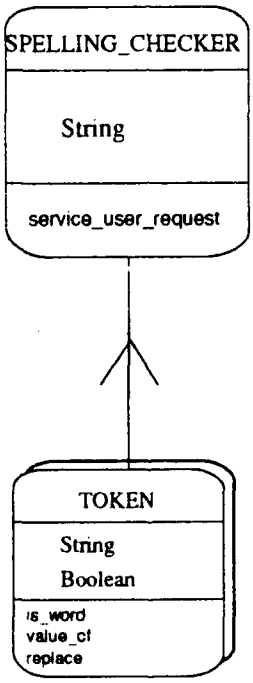


Figure 3-8. Has_Attribute Relationship Notation

The *has_attribute* relationship differs from the *has_part* relationship in that a *has_part* relationship denotes a structural decomposition while a *has_attribute* relationship denotes a semantic description of a class.

Has_association. The *has_association* relationship exists between two classes that only have the need to communicate. The classes involved in an association relationship in no way contribute state ranges to each other. Often, the *has_association* relationship is required in order to allow a class the ability to evaluate the value of another class.

Figure 3-9 shows the graphical representation of the *has_association* relationship between the *USER* class and the *SPELLING_CHECKER* class. This relationship is required since the *SPELLING_CHECKER* class and the *USER* class have the need to communicate with each other in order for the SCS to function. The *USER* does not provide any state range to the *SPELLING_CHECKER* and the *SPELLING_CHECKER* provides no state range to the *USER*.

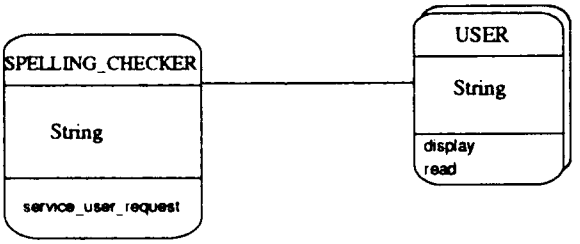


Figure 3-9. Has_Association Notation

Has_domain. The *has_domain* relationship exists only between a simple or complex class and a canonical class. This relationship describes the basis for the state range of that class without considering any of its parents, attributes, or parts. The total state range of any class is the combination of its *has_domain* relationships plus the state ranges of any parents, attributes, or parts. The state range is described by the range of values

the canonical class references. Canonical classes that have this relationship are listed in the middle area of the hidden icon. An example of this *has_domain* relationship can be seen in the *TOKEN* class (see Figure 3-10). This shows that the *TOKEN* class has the domain (state range) of the canonical class *String*, which was defined earlier. The diagram describes *TOKEN* as being a class whose individual instances are strings.

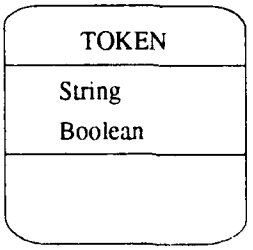


Figure 3-10.
Has_Domain Relationship
Notation

Multiplicity and Participation Restrictions. Multiplicity describes how many objects from each class are involved in the relationship, whereas participation describes whether or not the objects must participate in that relationship.

Multiplicity. The multiplicity part can be placed into four groups, *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many*. The *one-to-one* restriction placed on a relationship implies that one instance of a class has that relationship with one instance of another class, and the reverse must also be true. These restrictions can be illustrated using notation borrowed from (Shlaer and Mellor, 1988). Figure 3-11 shows how the restriction looks applied to two arbitrary sets S and W which represent classes. The instances (members) of each class (set) represent objects. In general, the left side of the restriction represents the class being defined and the right side depicts the class with which there is a relationship. The "one" means one instance of that class whereas "many" is more than one. Consequently, a

one-to-many restriction implies that any one instance of the first class is associated with more than one instance of the second class and many instances of second class is associated with one instance of the first class. A *many-to-one* restriction is just the reverse of the *one-to-many*.

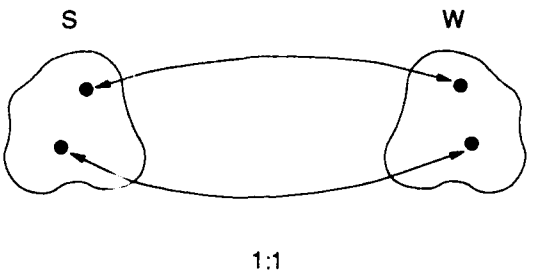


Figure 3-11. One-To-One Multiplicity

The last general multiplicity restriction is the *many-to-many* restriction which implies that many instances of the first class are associated with many instances of the second class and many instances of the second class are associated with many instances of the first class.

Participation. The other half of the restriction on relationships is the participation constraint. The "conditional" denotes that, under some conditions, an instance of one class may exist which does not have to take part in the relationship. A conditional on both sides of the restriction implies that instances from both classes can exist that do not participate in that relationship. Figure 3-12 shows the sets with conditional participation.

There are a total of sixteen different multiplicity and participation restrictions that may be placed on the various relationships, and certain relationships are subject to certain restrictions. All sixteen of these restrictions are described in Table 3-2.

The *has_domain* relationship is always restricted by the *one(conditional)-to-one* since any instance of a canonical class represents the mapping to only a single value at any time, any remaining values will not be part of relationship.

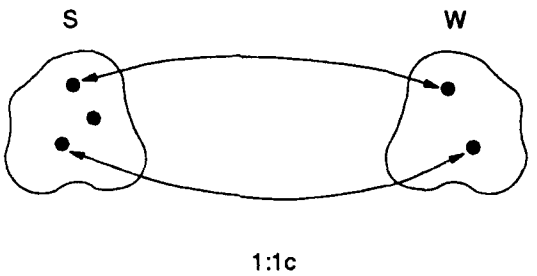
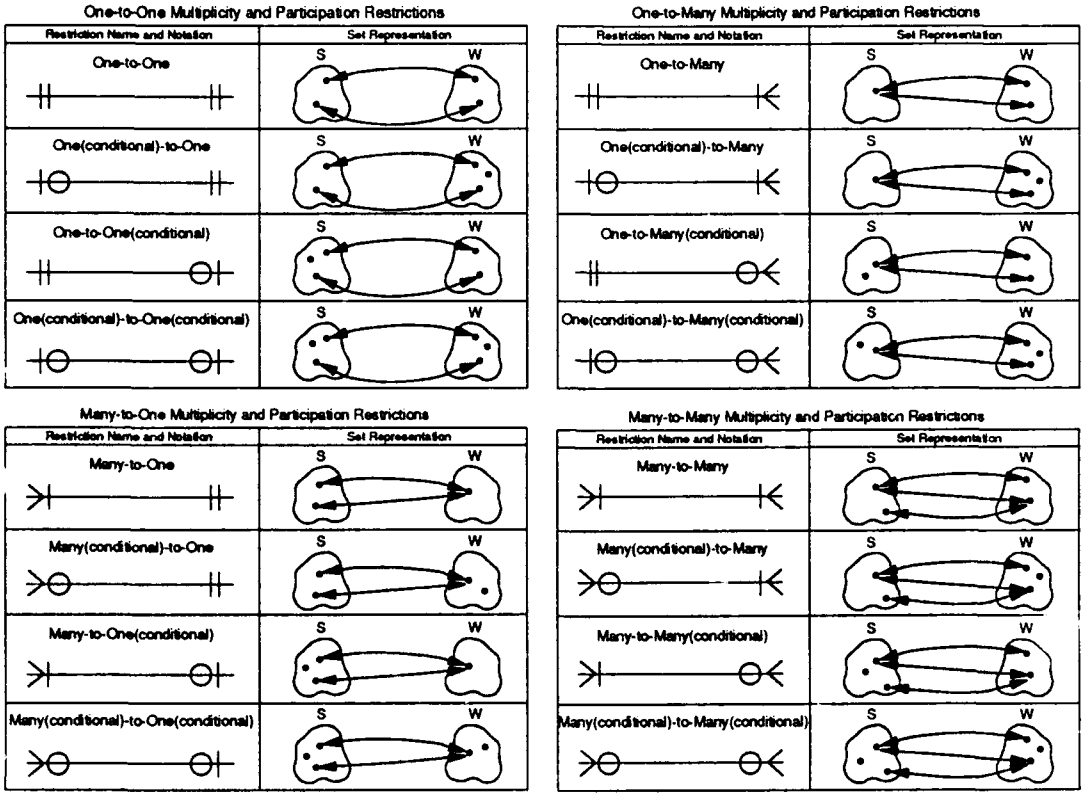


Figure 3-12. Conditional Participation

The *has_attribute* relationship is restricted to a *one(conditional)-to-one* or a *one-to-one* situation. That is, every instance of the class must have a corresponding attribute, but there may be instances of attributes that do not have a corresponding instance associated with it. The *has_part* and *has_association* can be any of the sixteen restrictions. By convention, any *has_association* relationship that is the result of a class being listed in the interface of another class is conditional on both ends of the relationship.

Relationships and restriction are best shown at the lowest level of abstraction possible. For example, the relationship between a class and the parent of another class may be different then the restrictions on the relationship between the class and the children of the pervious parent. Figure 3-13 illustrates the difference between showing a relationship at a higher level of abstraction (with a parent class) versus showing it at the lowest level (with a child class). The diagram shows that the relationship and restriction between the *SPELLING_CHECKER* class and *DOCUMENT* class is a *has_association* and *one(conditional)-to-many* whereas the

Table 3-2. Sixteen Multiplicity and Participation Restrictions



relationships and restrictions between the *DOCUMENT*'s children, *INPUT_DOC* and *OUTPUT_DOC*, and the *SPELLING_CHECKER* class are both *has_association* and *one(conditional)-to-one* instead of *one(conditional)-to-many.*

Behaviors. The lower area of the hidden icon lists all of the behaviors of this class. This list includes as a minimum all of the operations that were listed in the interface as capabilities. Behaviors that are listed only in the hidden view are behaviors classes in the hidden part (as we'll as the class itself) requests. An example of a behavior that would not be listed as a capability would be something like a garbage collecting routine which would only be callable from classes in the hidden part of this class. Figure 3-14 is the hidden view of the *TOKEN* class. The lower area of the hidden icon is where behaviors are listed. *TOKEN*'s behaviors are listed as *is_word*, *value_of*, and *replace.*

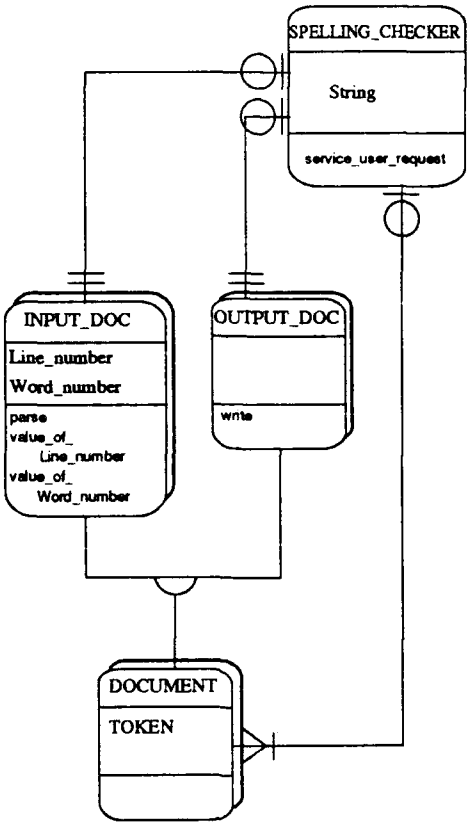


Figure 3-13. Defining Relationships at Different Levels of Abstraction

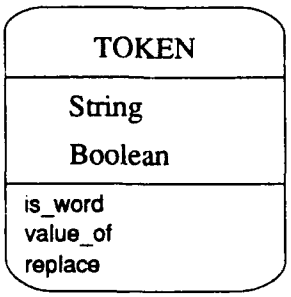


Figure 3-14. Behavior Notation

Formal Definition

Given the operational definition, the following is a formal definition of the object-oriented paradigm for requirements analysis:

(1) Definition. A Class is a 4-tuple <n,P,I,H> where

1. n is a unique identifier
2. P ▲ {parent_class ∈ Class}
3. I is a tuple describing the interface to n
4. H is a tuple describing the hidden part of n

(1.1) Definition. I is a 2-tuple <S,C> where

1. S ▲ {interface_class ∈ CLASS}
2. C ▲ {capability (n × interface_class)}

(1.2) Definition. H is a 2-tuple <M,B> where

1. M ▲ {(relationship (mult_part × n × mapping_class)}
2. B ▲ {behavior (n × mapping_class)}

and

A capability maps an instance of n to an instance of an interface_class.

A behavior maps an instance of n to an instance of a mapping_class.

and $C \subseteq B$

mapping_class ∈ Class and represents all of the classes associated with this class

thus $S \subseteq$ {mapping_class}

relationship ∈ {has_part, has_attribute, has_domain, has_association}

mult_part ∈ (1:1, 1:1c, 1c:1, 1c:1c, 1:M, 1:Mc, 1c:M, 1c:Mc, M:1,
M:1c, Mc:1, Mc:1c, M:M, M:Mc, Mc:M, Mc:Mc}

The multiplicity and participation restrictions are formally defined by the following definitions:

(2) Definition. One-to-one:

$1{:}1 \triangleq \{s{:}S; w{:}W \mid \forall s \in S, \exists! w \in W{:}(s,w) \wedge \forall w \in W, \exists! s \in S{:}(s,w)\}$

(3) Definition. One(conditional)-to-one:

$$1c{:}1 \triangleq \{s{:}S;w{:}W \mid \forall s \in S, \exists! w \in W{:}(s,w) \wedge (\forall w \in W, \forall s,s' \in S, ((s,w) \wedge (s',w)) \Rightarrow (s=s')\}$$

(4) Definition. One-to-one(conditional):

$$1{:}1c \triangleq \{s{:}S;w{:}W \mid \forall w \in W, \exists! s \in S{:}(s,w) \wedge (\forall s \in S, \forall w,w' \in W, ((s,w) \wedge (s,w')) \Rightarrow (w=w')\}$$

(5) Definition. One(conditional)-to-one(conditional):

$$1c{:}1c \triangleq \{s{:}S;w{:}W \mid (\forall s \in S, \forall w,w' \in W, ((s,w) \wedge (s,w')) \Rightarrow (w=w')) \wedge (\forall w \in W, \forall s,s' \in S, ((s,w) \wedge (s',w)) \Rightarrow (s=s'))\}$$

(6) Definition. One-to-many:

$$1{:}M \triangleq \{s{:}S;w{:}W \mid \forall s \in S, \exists w \in W{:}(s,w) \wedge \forall w \in W, \exists! s \in S{:}(s,w)\}$$

(7) Definition. One(conditional)-to-many:

$$1c{:}M \triangleq \{s{:}S;w{:}W \mid \forall s \in S, \exists w \in W{:}(s,w) \wedge (\forall w \in W, \forall s,s' \in S, ((s,w) \wedge (s',w)) \Rightarrow (s=s')\}$$

(8) Definition. One-to-many(conditional):

$$1{:}Mc \triangleq \{s{:}S;w{:}W \mid \forall w \in W, \exists! s \in S{:}(s,w)\}$$

(9) Definition. One(conditional)-to-many(conditional):

$$1c{:}Mc \triangleq \{s{:}S;w{:}W \mid (\forall w \in W, \forall s,s' \in S, ((s,w) \wedge (s',w)) \Rightarrow (s=s')\}$$

(10) Definition. Many-to-one:

$$M{:}1 \triangleq \{s{:}S;w{:}W \mid \forall s \in S, \exists! w \in W{:}(s,w) \wedge \forall w \in W, \exists s \in S{:}(s,w)\}$$

(11) Definition. Many(conditional)-to-one:

$$Mc{:}1 \triangleq \{s{:}S;w{:}W \mid \forall s \in S, \exists! w \in W{:}(s,w)\}$$

(12) Definition. Many-to-one(conditional):

$$M{:}1c \triangleq \{s{:}S;w{:}W \mid \forall w \in W, \exists s \in S{:}(s,w) \wedge (\forall s \in S, \forall w,w' \in W, ((s,w) \wedge (s,w')) \Rightarrow (w=w')\}$$

(13) Definition. Many(conditional)-to-one(conditional):

$$Mc{:}1c \triangleq \{s{:}S;w{:}W \mid (\forall s \in S, \forall w,w' \in W, ((s,w) \wedge (s,w')) \Rightarrow (w=w')\}$$

(14) Definition. Many-to-many:

$$M{:}M \triangleq \{s{:}S;w{:}W \mid \forall s \in S, \exists w \in W{:}(s,w) \wedge \forall w \in W, \exists s \in S{:}(s,w)\}$$

(15) Definition. Many(conditional)-to-many:

$$Mc{:}M \triangleq \{s{:}S;w{:}W \mid \forall s \in S, \exists w \in W{:}(s,w)\}$$

(16) Definition. Many-to-many(conditional):

$$M{:}Mc \triangleq \{s{:}S;w{:}W \mid \forall w \in W, \exists s \in S{:}(s,w)\}$$

(17) Definition. Many(conditional)-to-many(conditional):

$$Mc{:}Mc \triangleq \{s{:}S;w{:}W \mid \text{any subset of } S \times W\}$$

IV. Application

Chapter IV is the application of the formal model, developed in chapter III, to a sample problem to demonstrate the model's validity. The Air Traffic Control (ATC) simulation was chosen as the sample problem because it is sufficiently complex to demonstrate all of the aspects of the model. An initial object-oriented requirements analysis was performed on the ATC by Baum (Baum, 1991) and serves as the initial identification of classes and behaviors of the ATC system.

This chapter is divided into three parts. The first part is a description of the ATC simulation, the second part present the graphical representation of the ATC, and the third part presents the formal representation of the ATC.

Description of the ATC

The following is Baum's description of the ATC simulation (Baum, 1991):

> Air Traffic Control is a simulation which allows the user to play the part of an air traffic controller in charge of a 15x25 mile area from ground level to 9000 feet. In the area are 10 entry/exit fixes, 2 airports, and 2 navaids. During the simulation, 26 aircraft will become active, and it is the responsibility of the controller to safely direct these aircraft through his airspace.
>
> The controller communicates to the aircraft via the scope, issuing commands and status requests, receiving replies and reports, and noting the position of the aircraft on the map of the control space. The controller issues commands to change heading or altitude, to hold at a navaid, or clear for approach or landing. Each aircraft has a certain amount of fuel left, so the controller must see to it that the aircraft is dispositioned prior to fuel exhaustion. Also, the minimum separation rules must be followed, which state that no two aircraft may pass within three miles of each other at 1000 feet or less separation. The aircraft must enter and/or exit via one of the ten fixes. If an aircraft attempts to exit through a non-exit fix, a boundary error is generated. The controller may request a status report on each aircraft, which will display all information on the aircraft, including, fuel level, which is measured in minutes.

The player initially specifies the length of the simulation, which may be between 16 and 99 minutes. The same number of aircraft will appear for each run, so the shorter the simulation, the more challenging. The simulation terminates when all aircraft have been successfully dispositioned, the timer runs out, the player requests termination, or one of the three error conditions occurs:

- conflict error - separation rules were violated
- fuel exhaustion
- boundary error - the aircraft attempt to leave the control space via an unauthorized point.

Graphical Presentation of the ATC

The following section presents the graphical analysis of the ATC beginning with the highest level of abstraction and working toward the lowest level of abstraction.

Figure 4-1 represents the highest level of abstraction for the *ATC*. This interface icon of the *ATC* shows that in order to use the *ATC* system, a *USER* class and a canonical class *String* must also be present. It also describes the general capabilities of the system as *service_user_request*.

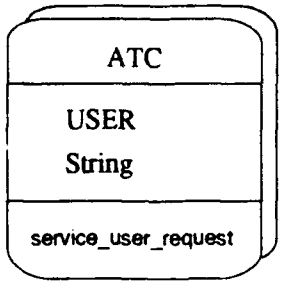


Figure 4-1. ATC Interface

Figure 4-2 represents the first level of detail of the ATC with this view of the hidden part of the *ATC* class. Now, the detail of the internal structure of the system begins to emerge with the addition of the *one-to-one has_part* relationship with *AIRSPACE* and the *one(conditional)-to-one(conditional) has_association* relationship with *COMMAND* which are

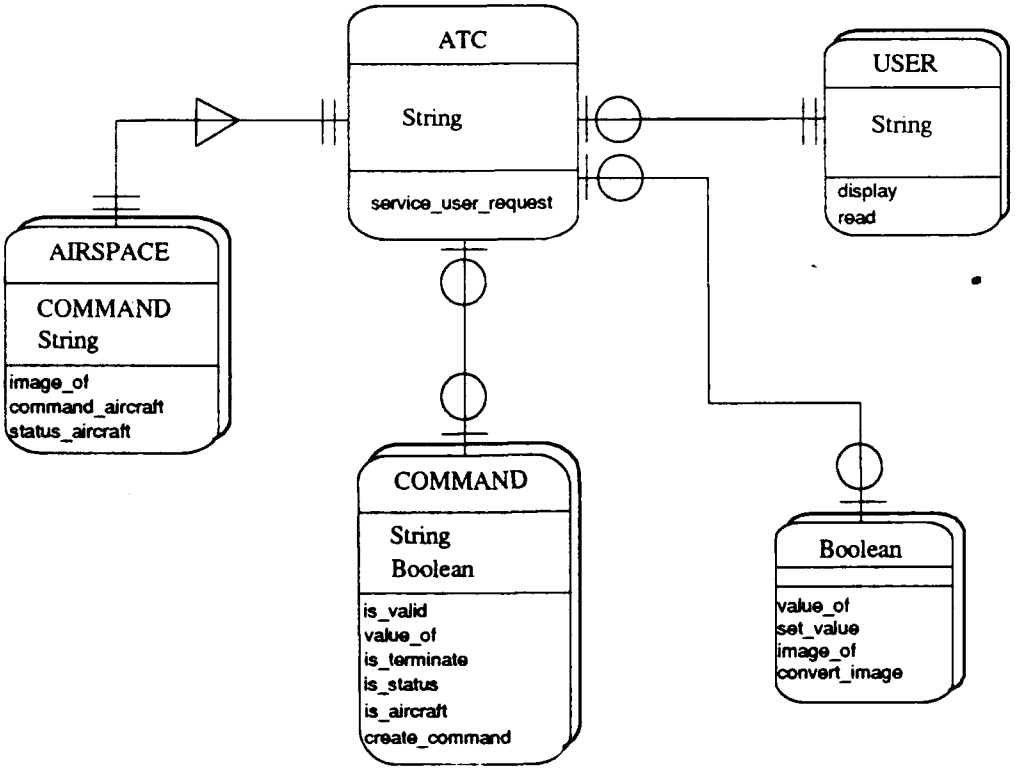


Figure 4-2. ATC Hidden View

both hidden from the *USER*. The *has_part* relationship with *AIRSPACE* implies that the *ATC* can be structurally decomposed into an *AIRSPACE* and the *one-to-one* restriction implies that if an instance of *ATC* exists, a corresponding instance of *AIRSPACE* must also exist. The *has_association* relationship with the canonical class *Boolean* is necessary since *Boolean* is listed in the interface of *COMMAND*. The *String* canonical class is left within the *ATC* icon

to show that *String* contributes to the state range of the *ATC* because of the *has_domain* relationship that exists. The rest of *ATC*'s state range is determined by *AIRSPACE* since the *AIRSPACE* is in a *has_part* relationship with *ATC*.

Figure 4-3 shows the hidden view of the *USER* class. The entire state range of *USER* is described by the canonical class *String*. The *one-to-one(conditional) has_association* relationship between the *USER* and *ATC* is provided to maintain continuity. The restriction states that if an *ATC* exists, there must be a *USER*. However, the reverse is not necessarily true. In other words, zero or one instances of *ATC* is associated with one instance of *USER*.

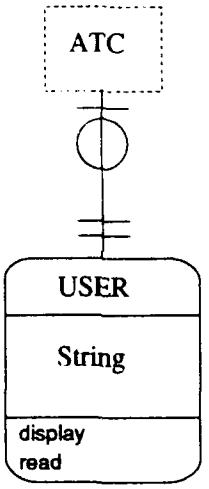


Figure 4-3. USER Hidden View

Figure 4-4 shows the hidden view of the *AIRSPACE* class. This view reveals that *AIRSPACE* can be decomposed into *has_part* relationships with *AIRCRAFT*, *FIX*, *NAVAID*, and *AIRPORT*. This view also shows the *has_association* relationships with *COMMAND*, *ID*, *POSITION*, and the canonicals *X_coordinate*, *Y_coordinate*, and *Z_coordinate*. There are also *has_domain* relationships with the canonicals *String* and *Boolean*. Once again, the relationship with *ATC* is shown for completeness.

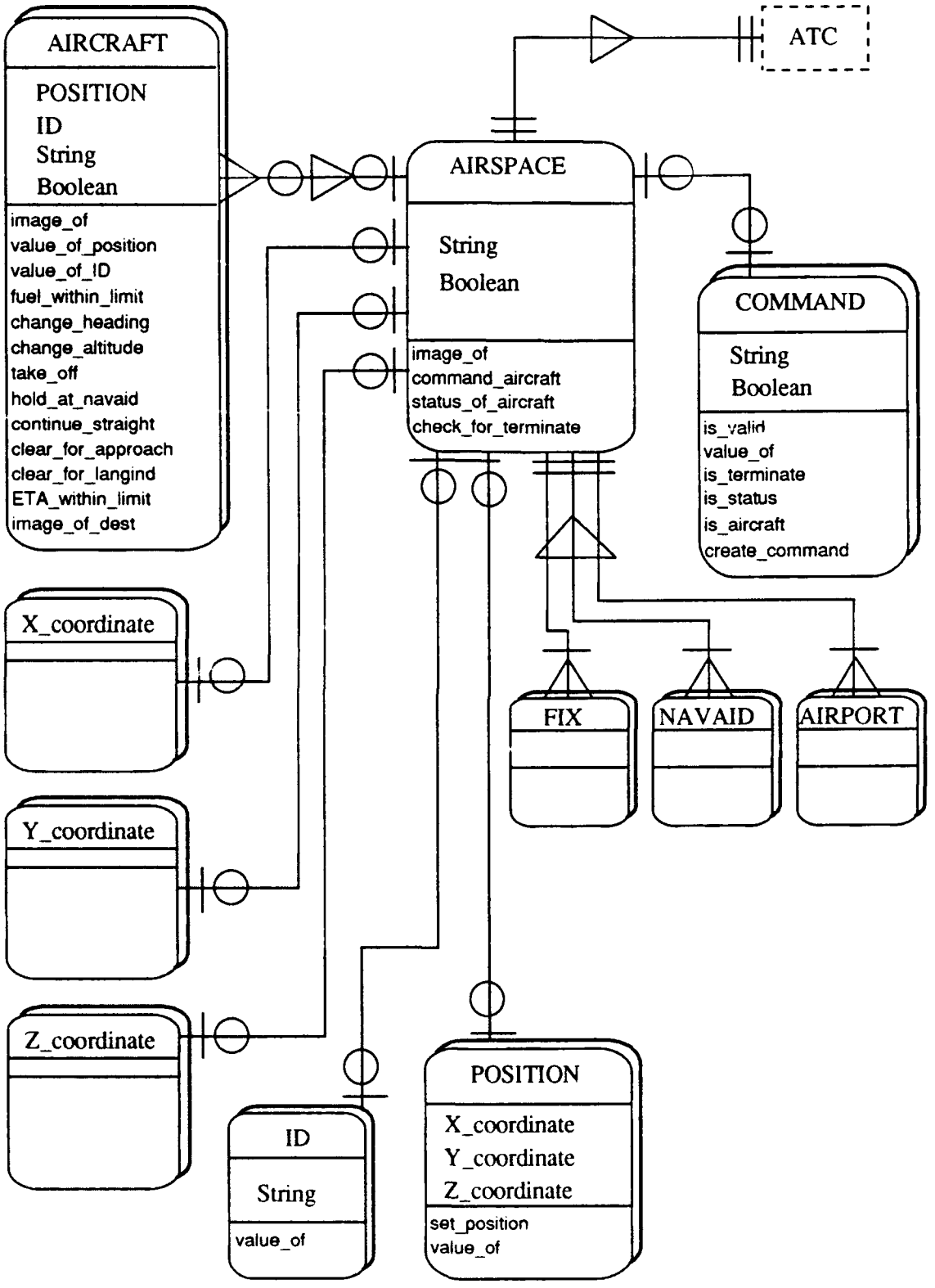


Figure 4-4. AIRSPACE Hidden View

It is also worth noting here that the behavior *check_for_terminate* is not considered to be a capability accessible from outside the *AIRSPACE* class and thus only listed in the hidden view and not the interface view.

The restriction on the *has_part* relationship between *AIRSPACE* and *AIRCRAFT* is show as *one(conditional)-to-many(conditional)*. This restriction states that there may be zero or more instances of *AIRCRAFT* (up to 26 see ATC description) associated with zero or one instances of *AIRSPACE*. Thus, an *AIRCRAFT* that had landed, exited, or had not yet become active would not have a relationship with an *AIRSPACE*, and an *AIRSPACE* that had no active *AIRCRAFT* would have no relationship with any instances of *AIRCRAFT*.

The ATC description calls for 10 fixes, 2 navaids, and 2 airports within the area. This corresponds directly to the *one-to-many* restrictions on the *has_part* relationships between *AIRSPACE* and the classes *FIX*, *NAVAID*, and *AIRPORT*.

The *one(conditional)-to-one(conditional) has_association* relationships with *ID* and *POSITION* are required since both *ID* and *POSITION* are listed in the interface of *AIRCRAFT*. Consequently, the relationships between *AIRSPACE* and the three coordinate canonical classes are as a result of the association with *POSITION*.

The *one(conditional)-to-one(conditional) has_association* with *COMMAND* is for communication of commands to *AIRSPACE*. Figure 4-5 shows the hidden view of *COMMAND*. *COMMAND*'s state range is determined by the canonicals *String* and *Boolean* and contributes none of that range to either *AIRSPACE* or *ATC*. The *one(conditional)-to-one(conditional)* restrictions between *COMMAND* and *ATC* and *AIRSPACE* are levied since zero or one instances of *COMMAND* can have a relationship with zero or one instances of *ATC* or *AIRSPACE*.

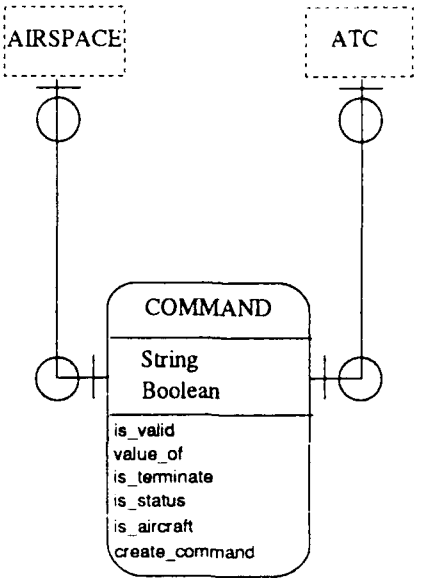


Figure 4-5. COMMAND Hidden View

Figure 4-6 diagrams the hidden view of the classes *FIX*, *NAVAID*, and *AIRPORT*. Keep in mind that these three classes inherit all of the *LANDMARK*'s interface and hidden parts. The parent *LANDMARK* is shown as an interface icon to maintain good modularity within the system.

Figure 4-7 presents the hidden view of the *LANDMARK* class. The *LANDMARK* class serves to provide a generalization of the types of physical entities found within the airspace of the ATC. Each instance of *LANDMARK* has an associated attribute of *POSITION* that describes where each *LANDMARK* exists within the airspace. The reason for the *one(conditional)-to-one* restriction on the relationship is because each instance of *LANDMARK* must have an associated instance of *POSITION*, but there may exist instances of *POSITION* not associated with a *LANDMARK*, namely those instances of *POSITION* that may be associated with the class *AIRCRAFT* since *POSITION* was also listed in *AIRCRAFT*'s interface.

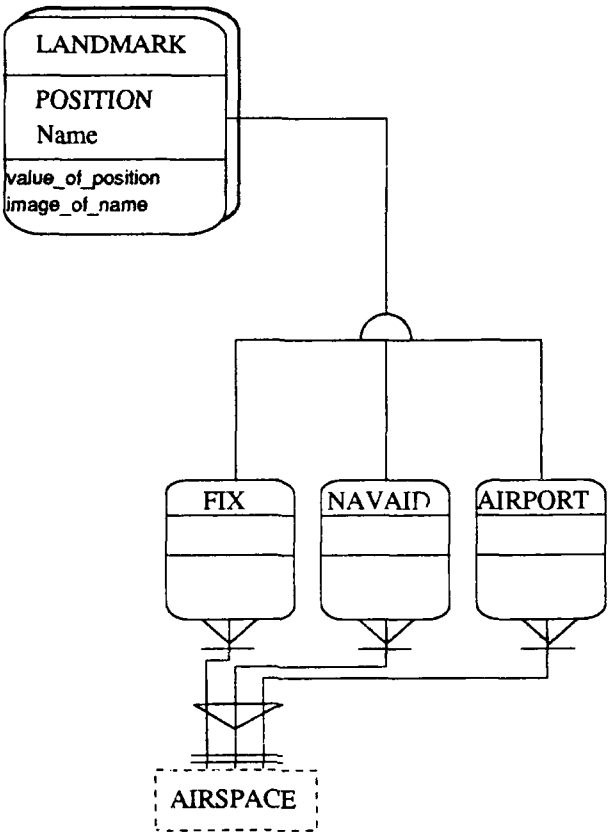


Figure 4-6. FIX, NAVAID, and AIRPORT Hidden View

Figure 4-8 provides the hidden view of the *AIRCRAFT* class. The *AIRCRAFT* class is described by several *has_attribute* relationships. Each instance of *AIRCRAFT* is described by an instance of *POSITION*, *FUEL*, *ID*, *HEADING*, *AIRCRAFT_TYPE*, and *FLIGHT_PLAN*. The only *has_attribute* relationship that is not *one-to-one* is the one with *POSITION* since there can be instances of *POSITION* associated with *LANDMARK* and not *AIRCRAFT*.

The *has_association* relationships with the coordinate canonical classes and the *Direction* canonical class are required since they are listed in the interface of *POSITION* and *HEADING* respectively.

Figure 4-9 presents the *FLIGHT_PLAN*'s hidden view. The *FLIGHT_PLAN* is made up of three parts: a *SOURCE*, a *DESTINATION*, and an *ETA*. The *has_association*

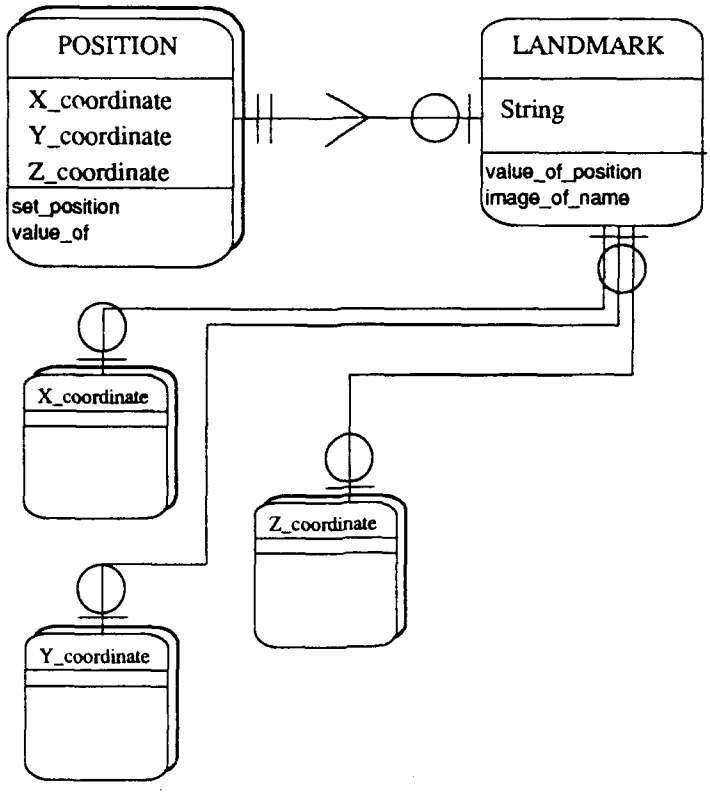


Figure 4-7. LANDMARK Hidden View

relationships are for convenience since both *Name* and *Number* are listed in the interface of the three *has_part* classes. The reason neither *Name* or *Number* had to be listed within the interface of *FLIGHT_PLAN* was because the capabilities were only passing an instance of the canonical class *String* in the form of *image_of_source*, *image_of_dest*, and *image_of_ETA* and not values.

Figure 4-10 is the hidden view of the classes *SOURCE*, *DESTINATION*, and *ETA*. These simple classes only have *has_domain* relationships with either *Name* or *Number*.

Figure 4-11 describes the hidden view of the *POSITION* class. Now, the specific relationships between *POSITION* and the coordinates are seen as *one-to-one has_parts*. Thus, every instance of *POSITION* must have an *X_coordinate*, a *Y_coordinate*, and a *Z_coordinate*; and every *X*, *Y*, and *Z_coordinate* must be associated with an instance of *POSITION*.

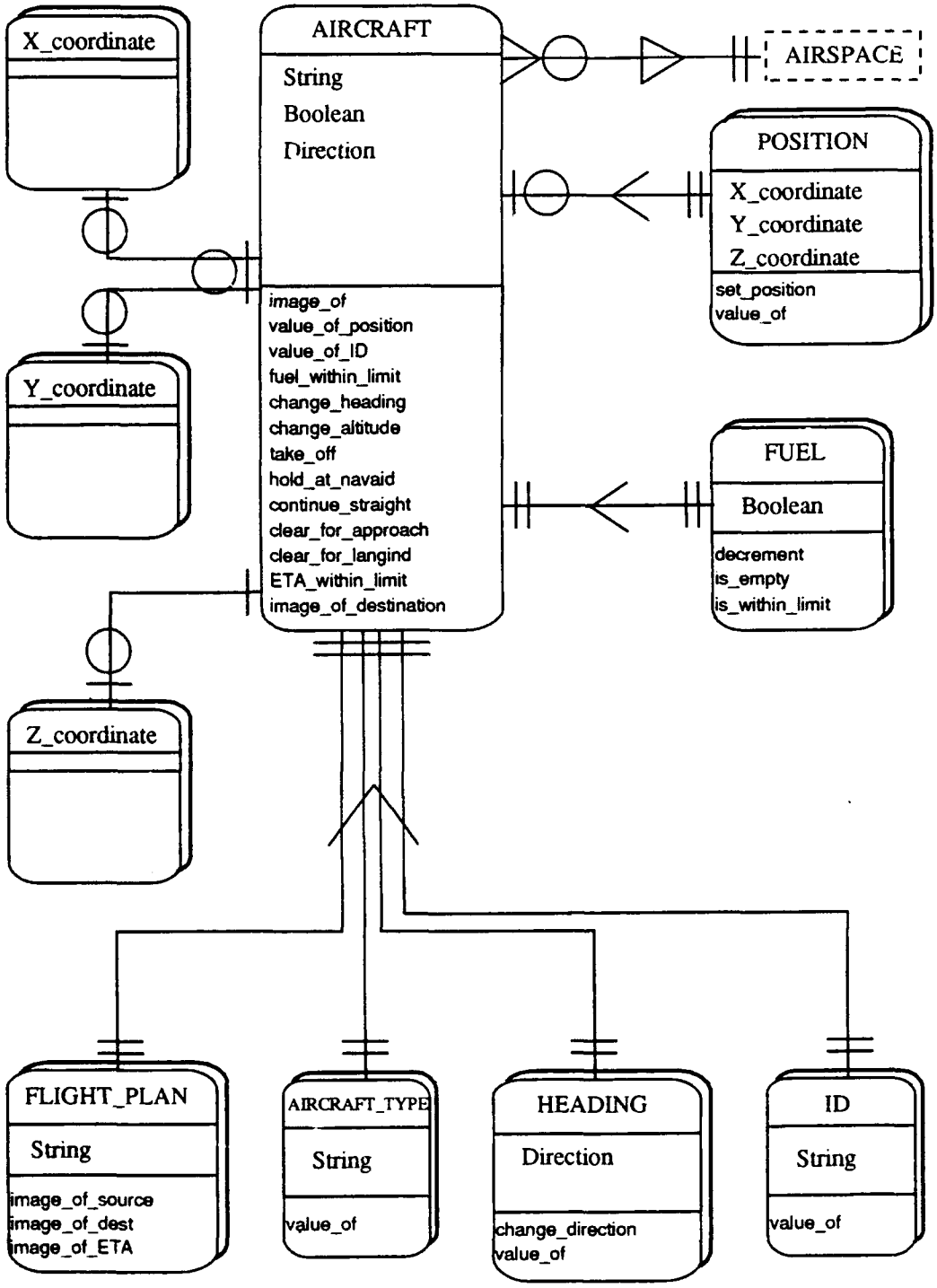


Figure 4-8. AIRCRAFT Hidden View

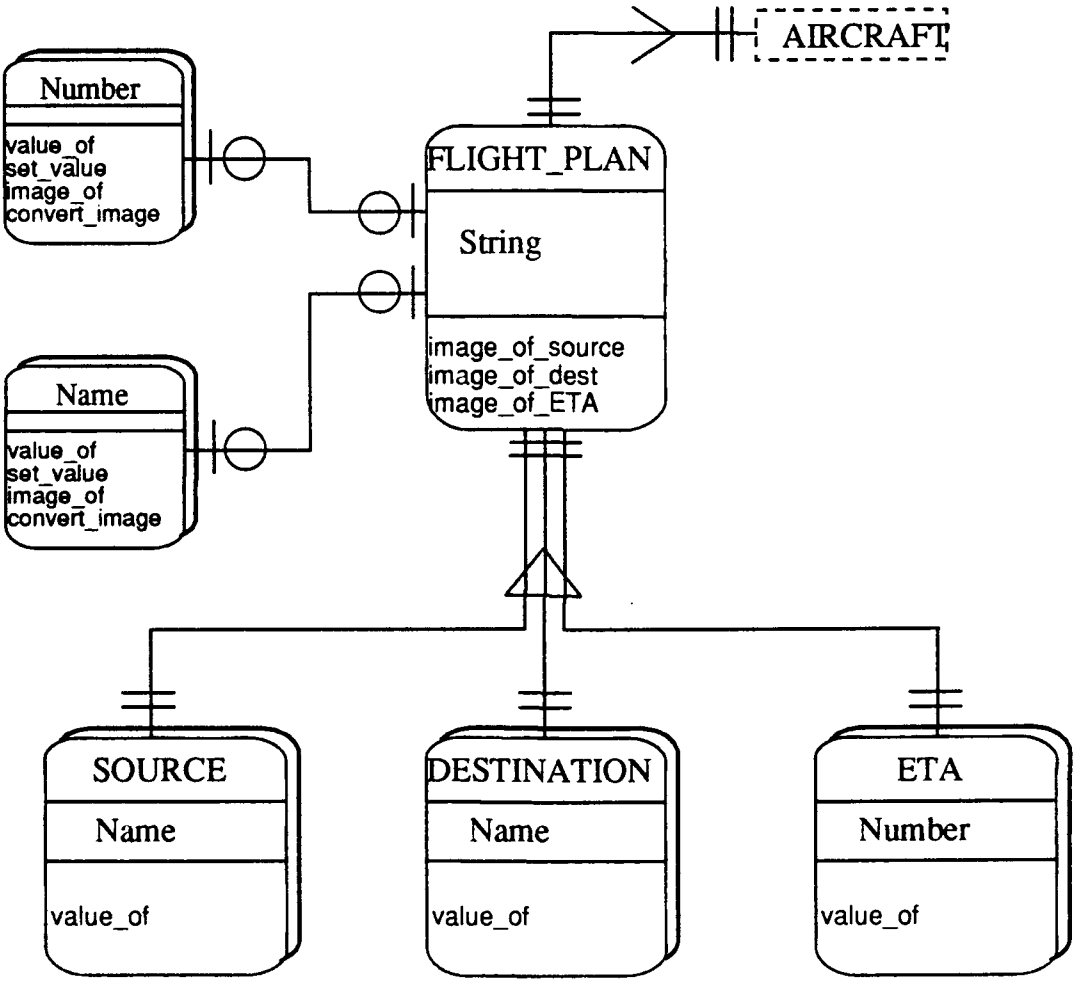


Figure 4-9. FLIGHT_PLAN Hidden View

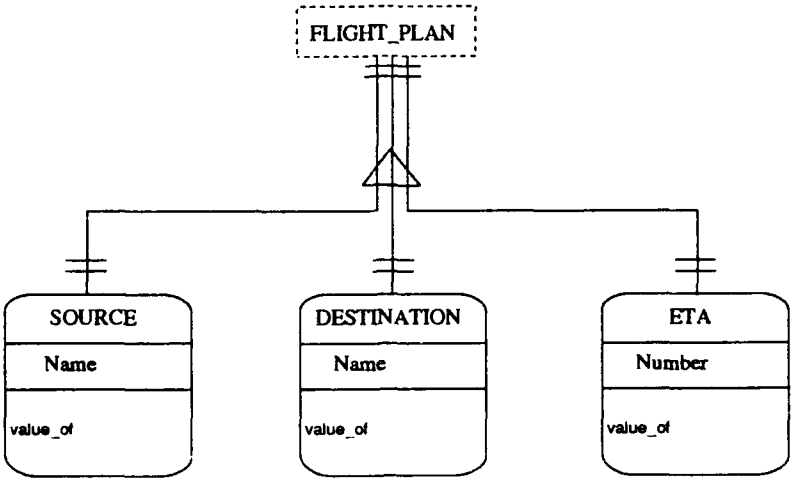


Figure 4-10. SOURCE, DESTINATION, and ETA Hidden View

Figure 4-12 presents the hidden view of the coordinate classes. The diagram shows that *X_coordinate*, *Y_coordinate*, and *Z_coordinate* are *a-kind-of Coordinate* class which is a canonical class. Since no interface classes or capabilities have been added to the interface of the child classes, they too can be considered as canonical classes.

The final diagram, Figure 4-13, describes the hidden view of the simple classes *AIRCRAFT_TYPE*, *HEADING*, *ID*, and *FUEL*. Nothing new has been added in the hidden view that was not present in the interface view.

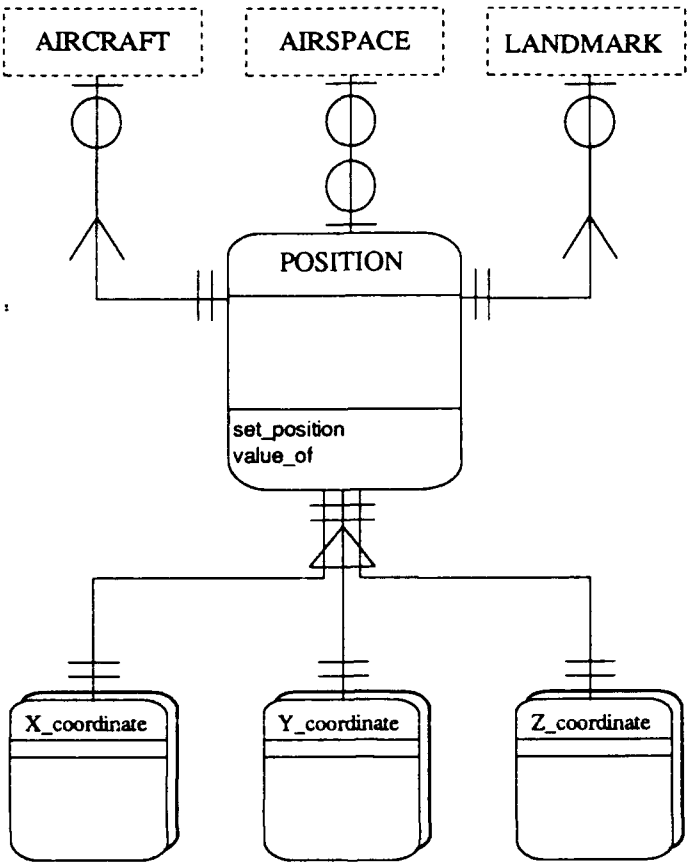


Figure 4-11. POSITION Hidden View

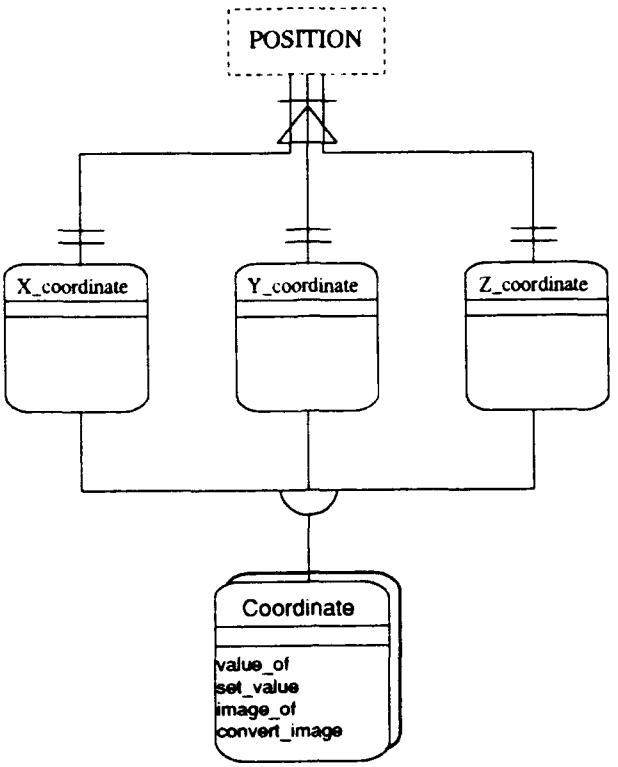


Figure 4-12. X_coordinate, Y_coordinate, and Z_coordinate Hidden Views

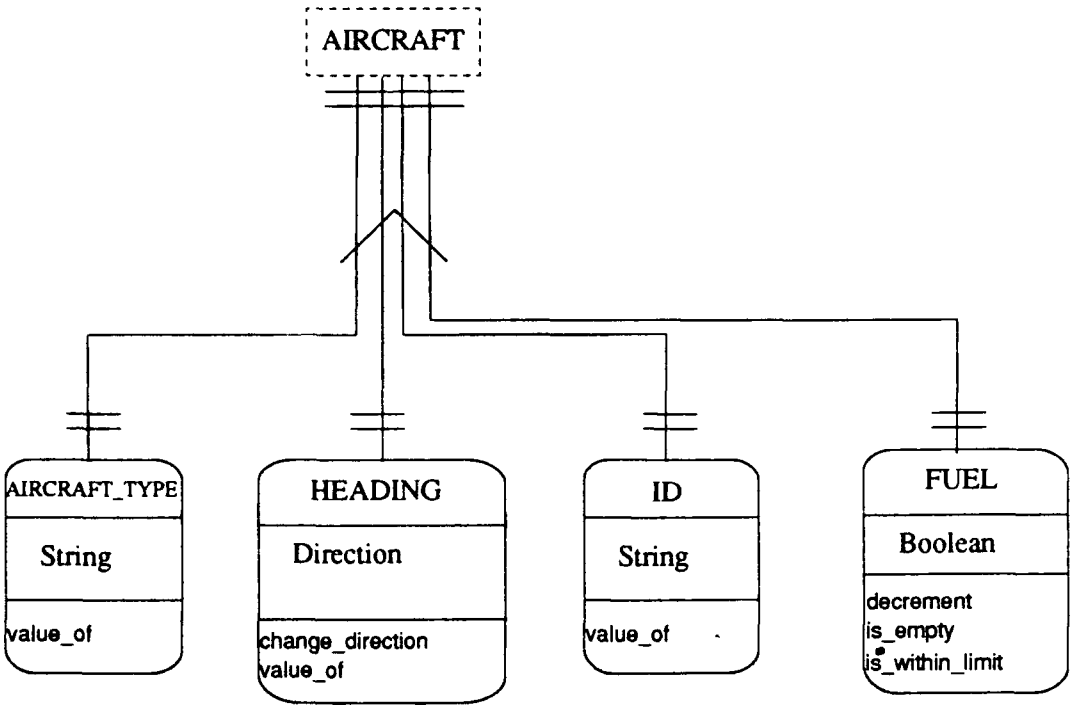


Figure 4-13. AIRCRAFT_TYPE, HEADING, ID, and FUEL Hidden Views

Formal Presentation of the ATC

There are 9 canonical classes, 11 simple classes, and 7 complex classes that make up the ATC system. The 9 canonical classes are: *String*, *Boolean*, *Name*, *Coordinate*, *X_coordinate*, *Y_coordinate*, *Z_coordinate*, *Direction*, and *Number*. The 11 simple classes are: *USER*, *COMMAND*, *POSITION*, *ID*, *FUEL*, *AIRCRAFT_TYPE*, *HEADING*, *FLIGHT_PLAN*, *SOURCE*, *DESTINATION*, and *ETA*. The 7 complex classes are: *ATC*, *AIRSPACE*, *AIRCRAFT*, *LANDMARK*, *FIX*, *NAVAID*, *AIRPORT*. The following section will formally define the above classes (classes will be listed in alphabetical order).

Formal Definitions of the ATC Classes.

AIRCRAFT

n = AIRCRAFT
P = {U}
I = <{POSITION, ID, String, Boolean},

{image_of(AIRCRAFT × String),
value_of_aircraft_position(AIRCRAFT × POSITION),
value_of_aircraft_ID(AIRCRAFT × ID),
fuel_within_limit(AIRCRAFT × Boolean),
change_heading(AIRCRAFT × AIRCRAFT),
change_altitude(AIRCRAFT × AIRCRAFT),
take_off(AIRCRAFT × AIRCRAFT),
hold_at_navaid(AIRCRAFT × AIRCRAFT),
continue_straight(AIRCRAFT × AIRCRAFT),
clear_for_approach(AIRCRAFT × AIRCRAFT),
clear_for_landing(AIRCRAFT × AIRCRAFT),
ETA_within_limit(AIRCRAFT × Boolean),
image_of_destination(AIRCRAFT × String)}>

H = <{has_domain(1c:1 × AIRCRAFT × Boolean),
has_domain(1c:1 × AIRCRAFT × String)},
has_attribute(1c:1 × AIRCRAFT × POSITION),
has_attribute(1:1 × AIRCRAFT × FLIGHT_PLAN),
has_attribute(1:1 × AIRCRAFT × ID),
has_attribute(1:1 × AIRCRAFT × FUEL),
has_attribute(1:1 × AIRCRAFT × CLASS),
has_attribute(1:1 × AIRCRAFT × HEADING),
has_association(1c:1c × AIRCRAFT × Direction),
has_association(1c:1c × AIRCRAFT × X_coordinate),
has_association(1c:1c × AIRCRAFT × Y_coordinate),
has_association(1c:1c × AIRCRAFT × Z_coordinate)},

{image_of(AIRCRAFT × String),
value_of_aircraft_position(AIRCRAFT × POSITION),
value_of_aircraft_ID(AIRCRAFT × ID),
fuel_within_limit(AIRCRAFT × Boolean),
change_heading(AIRCRAFT × AIRCRAFT),
change_altitude(AIRCRAFT × AIRCRAFT),
take_off(AIRCRAFT × AIRCRAFT),
hold_at_navaid(AIRCRAFT × AIRCRAFT),
continue_straight(AIRCRAFT × AIRCRAFT),
clear_for_approach(AIRCRAFT × AIRCRAFT),
clear_for_landing(AIRCRAFT × AIRCRAFT),

ETA_within_limit(AIRCRAFT × Boolean),
image_of_destination(AIRCRAFT × String)}>

AIRCRAFT_TYPE

n = AIRCRAFT_TYPE
P = {U}
I = <{String},

{value_of(AIRCRAFT_TYPE × String)}>

H = <{has_domain(1c:1 × AIRCRAFT_TYPE × String)},

{value_of(AIRCRAFT_TYPE × String)}>

AIRPORT

n = AIRPORT
P = {LANDMARK}
I = <{},{}>
H = <{},{}>

AIRSPACE

n = AIRSPACE
P = {U}
I = <{COMMAND, String},

{image_of(AIRSPACE × String),
command_aircraft(AIRSPACE × COMMAND),
status_aircraft(AIRSPACE × COMMAND)}>

H = <{has_domain(1c:1 × AIRSPACE × String),
has_domain(1c:1 × AIRSPACE × Boolean),
has_part(1c:Mc × AIRSPACE × AIRCRAFT),
has_part(1:M × AIRSPACE × FIX),
has_part(1:M × AIRSPACE × NAVAID),
has_part(1:M × AIRSPACE × AIRPORT),
has_association(1c:1 × AIRSPACE × COMMAND),
has_association(1c:1c × AIRSPACE × POSITION)},
has_association(1c:1c × AIRSPACE × X_coordinate),
has_association(1c:1c × AIRSPACE × Y_coordinate),
has_association(1c:1c × AIRSPACE × Z_coordinate),
has_association(1c:1c × AIRSPACE × ID)},

{image_of(AIRSPACE × String),
command_aircraft(AIRSPACE × COMMAND),

status_aircraft(AIRSPACE × COMMAND),
check_for_terminate(AIRSPACE × Boolean)}>

ATC

n = ATC
P = {U}
I = <{USER, String},

{service_user_request(ATC × String × USER)}>

H = <{has_domain(1c:1 × ATC × String),
has_part(1:1 × ATC × AIRSPACE)
has_association(1c:1c × ATC × Boolean),
has_association(1c:1c × ATC × COMMAND),
has_association(1c:1 × ATC × USER)},

{service_user_request(ATC × String × USER)}>

Boolean ≙ {true, false}

COMMAND

n = COMMAND
P = {U}
I = <{String, Boolean},

{is_valid(COMMAND × Boolean),
value_of(COMMAND × String),
is_terminate(COMMAND × Boolean),
is_status(COMMAND × Boolean),
is_aircraft(COMMAND × Boolean),
create_command(COMMAND × String)}>

H = <{has_domain(1c:1 × COMMAND × String),
has_domain(1c:1 × COMMAND × Boolean)},

{is_valid(COMMAND × Boolean),
value_of(COMMAND × String),
is_terminate(COMMAND × Boolean),
is_status(COMMAND × Boolean),
is_aircraft(COMMAND × Boolean),
create_command(COMMAND × String)}>

Coordinate ≙ $\{c \in \mathbf{N} \mid 0 \leq c \leq \infty\}$

DESTINATION

n = DESTINATION
P = {U}

I = <{Name},

{value_of(DESTINATION × Name)}>

H = <{has_domain(1c:1 × DESTINATION × Name)},

{value_of(DESTINATION × Name)}>

Direction ≙ {N, NE, E, SE, S, SW, W, NW}

ETA

n = ETA
P = {U}
I = <{Number},

{value_of(ETA × Number)}>

H = <{has_domain(1c:1 × ETA × Number)},

{value_of(ETA × Number)}>

FIX

n = FIX
P = {LANDMARK}
I = <{},{}>
H = <{},{}>

FLIGHT_PLAN

n = FLIGHT_PLAN
P = {U}
I = <{String},

{image_of_source(FLIGHT_PLAN × String),
image_of_destination(FLIGHT_PLAN × String),
image_of_ETA(FLIGHT_PLAN × String)}>

H = <{has_domain(1c:1 × FLIGHT_PLAN × String),
has_attribute(1:1 × FLIGHT_PLAN × SOURCE),
has_attribute(1:1 × FLIGHT_PLAN × DESTINATION),

has_attribute(1:1 × FLIGHT_PLAN × ETA),
has_association(1c:1c FLIGHT_PLAN × Name),
has_association(1c:1c FLIGHT_PLAN × Number)}>

{image_of_source(FLIGHT_PLAN × String),
image_of_destination(FLIGHT_PLAN × String),
image_of_ETA(FLIGHT_PLAN × String)}>

FUEL

n = FUEL
P = {U}
I = <{Boolean},

{decrement(FUEL × FUEL),
is_empty(FUEL × Boolean),
is_within_limit(FUEL × Boolean)}>

H = <{has_domain(1c:1 × FUEL × Boolean)},

{decrement(FUEL × FUEL),
is_empty(FUEL × Boolean),
is_within_limit(FUEL × Boolean)}>

HEADING

n = HEADING
P = {U}
I = <{Direction},

{change_direction(HEADING × HEADING),
value_of(HEADING × Direction)}>

H = <{has_domain(1c:1 × HEADING × Direction)},

{change_direction(HEADING × HEADING),
value_of(HEADING × Direction)}>

ID

n = ID
P = {U}
I = <{String},

{value_of(ID × String)}>

H = <{has_domain(1c:1 × ID × String)},

{value_of(ID × String)}>

LANDMARK

n = LANDMARK
P = {U}
I = <{POSITION, String},

{value_of_position(LANDMARK × POSITION),
image_of(LANDMARK × String)}>

H = <{has_domain(1c:1 × LANDMARK × String),
has_domain(1c:1 × LANDMARK × Name),
has_attribute(1c:1 × LANDMARK × POSITION),
has_association(1c:1c × LANDMARK × X_coordinate),
has_association(1c:1c × LANDMARK × Y_coordinate),
has_association(1c:1c × LANDMARK × Z_coordinate)},

{value_of_position(LANDMARK × POSITION),
image_of(LANDMARK × String)}>

Name ≙ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *, #, %}

NAVAID

n = NAVAID
P = {LANDMARK}
I = <{},{}>
H = <{},{}>

Number ≙ {n ∈ **N** | 1 ≤ n ≤ 10}

POSITION

n = POSITION
P = {U}
I = <{X_coordinate, Y_coordinate, Z_coordinate},

{set_position(POSITION × X_coordinate × Y_coordinate ×
Z_coordinate},
value_of(POSITION × X_coordinate × Y_coordinate ×
Z_coordinate)}>

H = <{has_part(1:1 × POSITION × X_coordinate),
has_part(1:1 × POSITION × Y_coordinate),

has_part(1:1 × POSITION × Z_coordinate)}

{set_position(POSITION × X_coordinate × Y_coordinate × Z_coordinate},
value_of(POSITION × X_coordinate × Y_coordinate × Z_coordinate)}>

SOURCE

n = SOURCE
P = {U}
I = <{Name},

{value_of(SOURCE × Name)}>

H = <{has_domain(1c:1 × SOURCE × Name)},

{value_of(SOURCE × Name)}>

String ≜ {ASCII_alphabet}$^{+}$

USER

n = USER
P = {U}
I = <{String},

{display(USER × String),
read(USER × String)}>

H = <{has_domain(1c:1 × USER × String)},

{display(USER × String),
read(USER × String)}>

X_coordinate ≜ {x ∈ *Coordinate* | 1 ≤ x ≤ x_max}

Y_coordinate ≜ {y ∈ *Coordinate* | 1 ≤ y ≤ y_max}

Z_coordinate ≜ {z ∈ *Coordinate* | 0 ≤ z ≤ . .nax}

Summary of Application

The preceding specification of the ATC simulation provides sufficient detail about the system to begin the design phase. The ATC has been leveled from a single interface description (highest level of abstraction) down to the simple and canonical classes (lowest level of abstraction) which describe the state range of the entire system. This specification is considered to be an accurate representation of an OORA of the ATC simulation.

V. *Conclusions and Recommendations*

This final chapter summarizes the research presented in this thesis. It also provides a list of conclusions that have been reached as a result of the research, and provides some recommendations for follow-on research for the formalism developed.

Summary

The literature surveyed in chapter II for methods of doing OORA revealed informal based methods by Bailin, Shlaer and Mellor, Booch, and Coad and Yourdon. Formal based methods included Bralick, Z, and REFINE. However, none were found to be adequate for producing an OORA. The informal methods varied in approach from a combination of structured and object-oriented to pure object-oriented with everything on the same plane of abstraction. Of the formal methods, Bralick's was the only one to claim to be object-oriented. However, it did not meet the criteria for being a formal definition. The remaining two formal based methods are broad based in their application and not specifically applied to the O-O paradigm.

A formal method for OORA was developed in chapter III. The formalism presented a combination of graphical representation, using the Spelling Checker System as examples, and formal definitions. The formalism revolves around the concept of a class. Formally, a class is a 4-tuple consisting of a unique name, a set of parents, an interface part and a hidden part.

The formalism was demonstrated by applying it to the ATC simulation. Both the graphical and formal presentations proved to be successful for conducting an OORA. The resulting specification of the ATC was sufficient to begin the design phase.

Conclusions

Several conclusions can be drawn from the research presented in this thesis.

1) An OORA can successfully be formalized as a combination of graphical notations and formal definitions based on set and relation theory.

2) Classes are more appropriate than objects for building a requirements analysis model since a class is a template whereas an object is specific (an object may be more appropriate for a design model).

3) No adequate formal based methods currently exist upon which to build a object-oriented formal specification.

4) There are specific relationships that exist between classes; namely *has_part*, *has_attribute*, *has_association*, *has_domain*, and *a-kind-of*. These relationships determine the structure and state range of the entire system under analysis.

5) There are 16 multiplicity and participation restrictions that are placed on the relationships between classes (except the a-kind-of relationship). These restrictions describe the number of instances from each class participate in the relationship.

6) A graphical representation alone was not sufficient to provide an adequate specification. The graphic does provides excellent support for developing the formal specification.

7) Contrary to many of the object-oriented methods reviewed to date, an attribute is nothing more than another class. The relationship between a class that serves to describe another class is a *has_attribute* relationship.

8) Three levels of class complexity were discovered: canonical, simple, complex. Canonical classes provided mapping to values, simple classes had only canonical classes in its interface, and complex classes had other simple or complex classes in its interface.

9) At analysis time, there are few differences between a class's list of capabilities and its list of behaviors.

In addition to the previous conclusions, several heuristics were drawn out of the research.

1) When specifying relationships and restrictions between classes, and one of those classes is the parent of another class, ensure that the relationships and restrictions are valid for both the parent and any children that would inherit those relationships and restrictions, otherwise, define the relationships from the child class only.

2) Certain relationships are constrained by certain restrictions. The *has_domain* is always restricted by the *one(conditional)-to-one*, the *has_attribute* is always restricted to either *one(conditional)-to-one* or *one-to-one*. By convention,

has_association relationships that are a result of another relationship are conditional on both sides.

3) The more interface classes a class declares, the more complex the overall system becomes. Every class listed in a class's interface must also have some relationship with every client of that class.

4) An OORA should be leveled by means of the interface view and hidden view.

Recommendations

As a result of the research, several follow-on topics are recommended.

1) The formal definition presented in this thesis should be formally proven. This could be accomplished by proving it to be isomorphic to a turing machine. The effect of this would be to further validate the formalism.

2) Develop complexity metrics based on the interface descriptions of the classes. To do this would require several applications of the formalism to partition a number of systems into the three levels of class complexity. Once done, expert opinion would be needed to quantify the actual levels of system complexity. Results from the formalism and opinion poll could then be correlated to determine any relationships between complexity and interface description.

3) The formal specification, developed as a result of applying this formal model, could be used to model the dynamics of a given system by applying set and

relations operations on the formal specification itself. The result could be similar to a prototype of the system.

4) Take this formal model to the next step of the software development process and apply it to OOD. At this level, objects would have to be introduced along with their associated state.

Remarks

This research was undertaken to promote the use of formal definitions in the software development process. It is believed that through the use of formalism, like the one developed in this thesis, the software development process might be a little less ad hoc. Systems developed using these formalism will be better understood earlier on in the program and easier to maintain once fielded.

Appendix: Spelling Checker System

Description of the Spelling Checker System

The following Spelling Checker System description was presented by (Umphress, 1990):

> The spelling checker will parse an input document extracting one word at a time. (As it parses the document, it displays the current document line number and word number on the terminal.) If the word is a "non-word", the spelling checker passes it directly to the output document. Otherwise, the spelling checker consults the main and temporary dictionaries. If either dictionary contains the word, the word is written on the output document.
>
> If neither dictionary contains the word, the spelling checker allows the operator to either:
>
> (1) replace the word with a word typed in from the terminal,
> (2) add the word to the temporary dictionary,
> (3) add the word to the main dictionary, or
> (4) request the program to display like-spelled words in the main and temporary dictionaries.
>
> The word entered or accepted by the user is written to the output document.
>
> The space available in the main and temporary dictionaries will be continuously displayed on the terminal. An error message will be displayed if the user tries to add a word to a full dictionary.

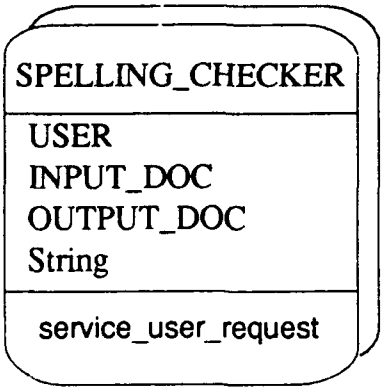


Figure A-1. SPELLING_CHECKER Interface

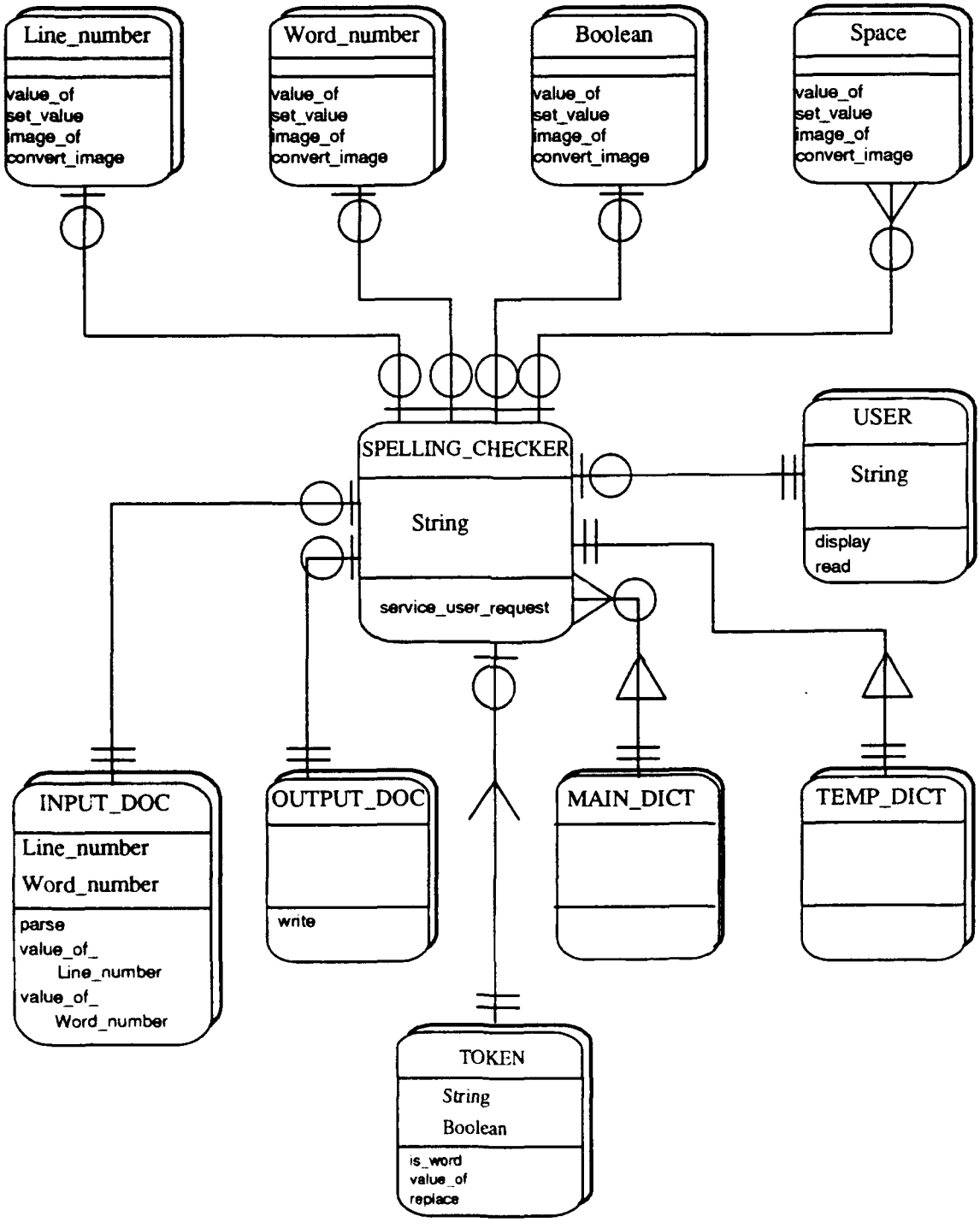


Figure A-2. SPELLING_CHECKER Hidden View

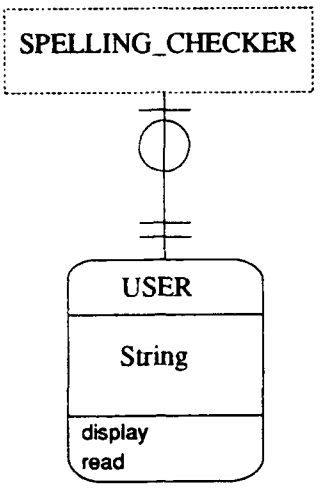


Figure A-3. USER Hidden View

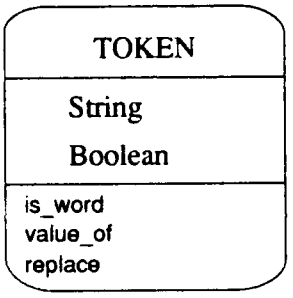


Figure A-4. TOKEN Hidden View

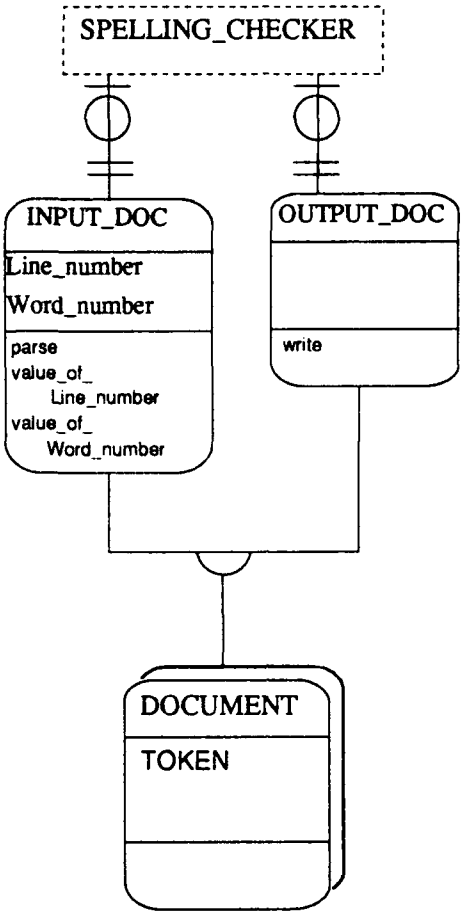


Figure A-5. INPUT_DOC and OUTPUT_DOC Hidden Views

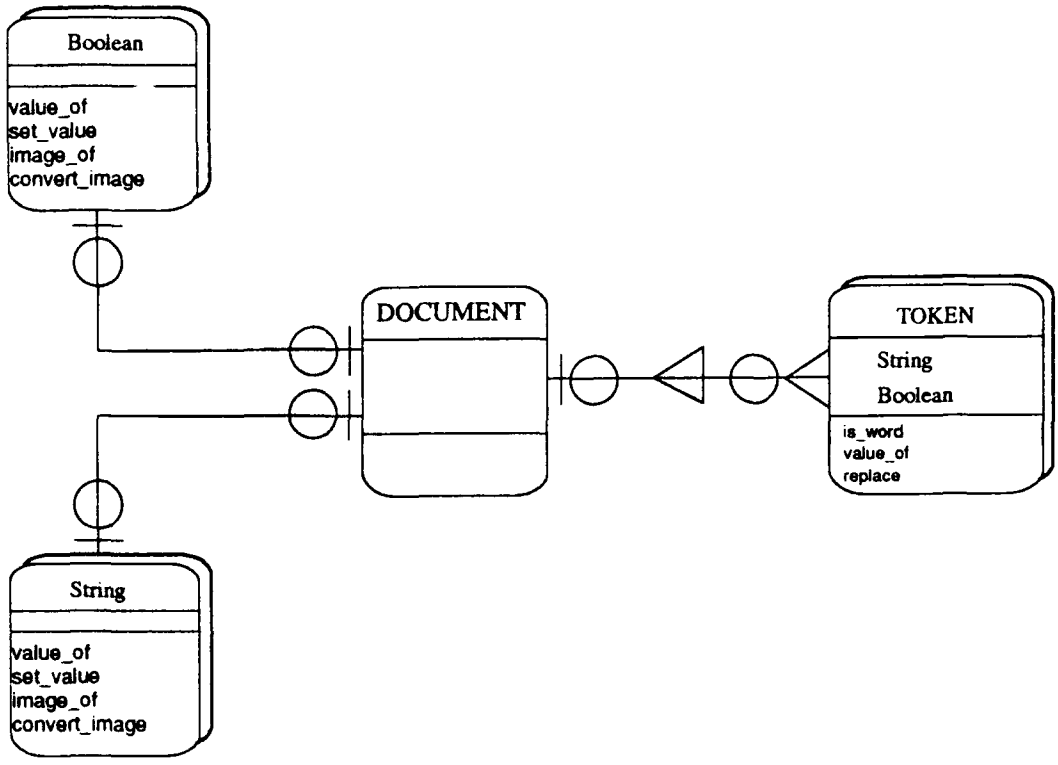


Figure A-6. DOCUMENT Hidden View

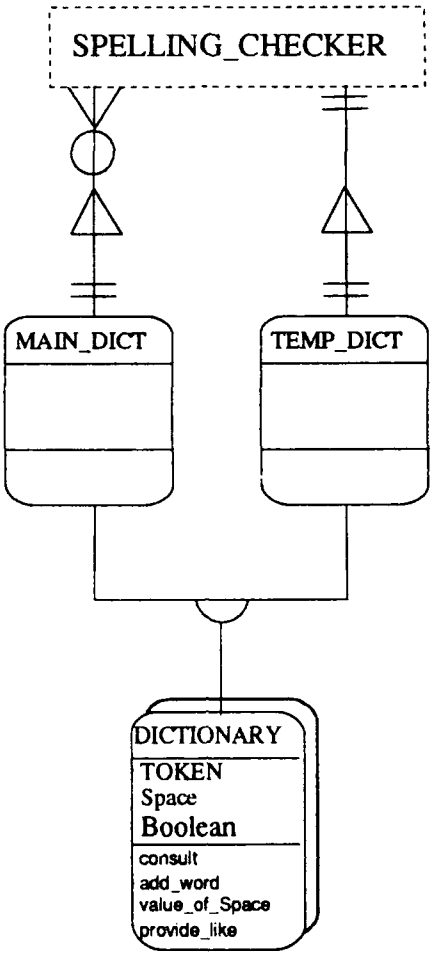


Figure A-7. MAIN_DICT and TEMP_DICT Hidden Views

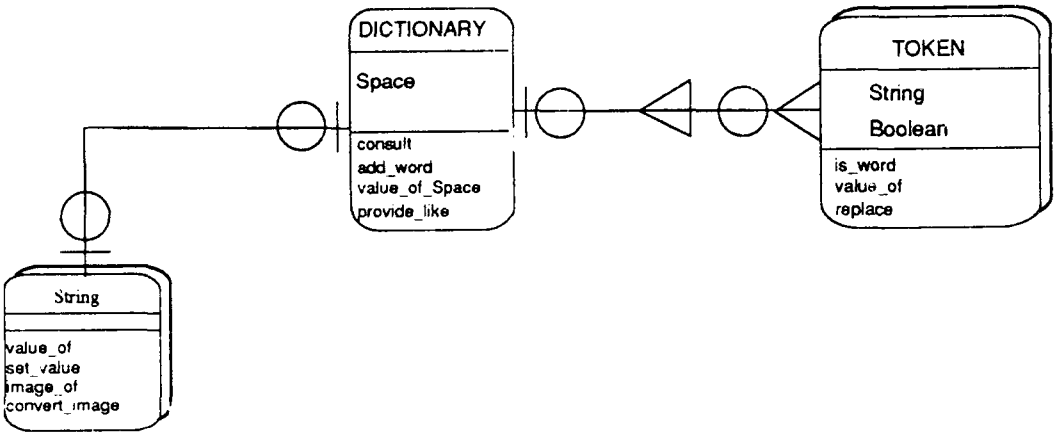


Figure A-8. DICTIONARY Hidden View

Formal Presentation of the SCS

Definition of the Spelling Checker System Classes (in alphabetical order).

Boolean ≙ {true, false}

DICTIONARY Class Definition

n = DICTIONARY
P = {U}
I = <{TOKEN, Space, Boolean},

{(consult (DICTIONARY × TOKEN × Boolean)),
(add_word (DICTIONARY × TOKEN × DICTIONARY)),
(value_of (DICTIONARY × Space)),
(provide_like (DICTIONARY × TOKEN))}>

H = <{(has_domain (1c:1 × DICTIONARY × Space)),
(has_domain (1c:1 × DICTIONARY × Boolean)),
(has_part (1c:Mc × DICTIONARY × TOKEN)),
(has_association (1c:1c DICTIONARY × String))}

{(consult (DICTIONARY × TOKEN × Boolean)),
(add_word (DICTIONARY × TOKEN × DICTIONARY)),
(value_of (DICTIONARY × Space)),
(provide_like (DICTIONARY × TOKEN))}>

DOCUMENT Class Definition

n = DOCUMENT
P = {U}
I = <{TOKEN}, {}>

H = <{(has_part (1c:Mc × DOCUMENT × TOKEN)),
(has_association (1c:1c × DOCUMENT × String))
(has_association (1c:1c × DOCUMENT × Boolean))}, {}>

INPUT_DOC Class Definition

n = INPUT_DOC
P = {DOCUMENT}
I = <{Line_number, Word_number},

{(parse (INPUT_DOC × TOKEN)),
(value_of (INPUT_DOC × Line_number)),
(value_of (INPUT_DOC × Word_number))}>

H = <{(has_domain (1c:1 × INPUT_DOC × Line_number)),
(has_domain (1c:1 × INPUT_DOC × Word_number))},

{(parse (INPUT_DOC × TOKEN)),
(value_of (INPUT_DOC × Line_number)),
(value_of (INPUT_DOC × Word_number))}>

Line_number ≜ {l ∈ **N** | 0 ≤ l ≤ l_max}

MAIN_DICT Class Definition

n = MAIN_DICT
P = {DICTIONARY}
I = <{},{}>
H = <{},{}>

OUTPUT_DOC Class Definition

n = OUTPUT_DOC
P = {DOCUMENT}
I = <{}, {write (OUTPUT_DOC × TOKEN)}>
H = <{}, {write (OUTPUT_DOC × TOKEN)}>

Space ≜ {s ∈ **R** | 0.0 ≤ s ≤ 1.0}

SPELLING_CHECKER Class Definition

n = SPELLING_CHECKER
P = {U}
I = <{USER, INPUT_DOC, OUTPUT_DOC, String},

{service_user_request (SPELLING_CHECKER × String × USER ×
INPUT_DOC × OUTPUT_DOC)}>

H = <{(has_domain (1c:1 × SPELLING_CHECKER × String)),
(has_part (Mc:1 × SPELLING_CHECKER × MAIN_DICT)),
(has_part (1:1 × SPELLING_CHECKER × TEMP_DICT)),
(has_attribute (1c:1 × SPELLING_CHECKER × TOKEN)),
(has_association (1c:1 × SPELLING_CHECKER × USER)),
(has_association (1c:1 × SPELLING_CHECKER × INPUT_DOC)),
(has_association (1c:1 × SPELLING_CHECKER × OUTPUT_DOC))
(has_association (1c:1c × SPELLING_CHECKER × Line_number))
(has_association (1c:1c × SPELLING_CHECKER × Word_number))
(has_association (1c:1c × SPELLING_CHECKER × Boolean))
(has_association (1c:Mc × SPELLING_CHECKER × Space))},

{service_user_request (SPELLING_CHECKER × String × USER ×
INPUT_DOC × OUTPUT_DOC)}>

String ≙ {ASCII_alphabet}$^{+}$

TEMP_DICT Class Definition

n = TEMP_DICT
P = {DICTIONARY}
I = <{},{}>
H = <{},{}>

TOKEN Class Definition

n = TOKEN
P = {U}
I = <{String, Boolean},

{(is_word (TOKEN × Boolean)),
(value_of (TOKEN × String)),
(replace (TOKEN × TOKEN))}>

H = <{(has_domain (1c:1 × TOKEN × String)),
(has_domain (1c:1 × TOKEN × Boolean))},

{is_word (TOKEN × Boolean),
value_of (TOKEN × String),
replace (TOKEN × TOKEN)}>

USER Class Definition

n = USER
P = {U}
I = <{String},

{display (USER × String),
read (USER × String)}>

H = <{(has_domain (1c:1 × USER × String))},

{display (USER × String),
read (USER × String)}>

Word_number ≙ {w ∈ **N** | 0 ≤ w ≤ w_max}

Bailin, Sidney C. 1989. "An Object-Oriented Requirements Specification Method," Communications of the ACM, 32, 5:608-623.

Bailor, Paul. 1991. Class handout distributed in CSCE 595, Software Generation and Maintenance. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH.

Baum, K. D. and D. Umphress. 1991 (In press). "Determining Concurrency in Object-Oriented Real-Time Design." Ada Letters.

Booch, Grady. 1987. Software Components with Ada: Structures, Tools, and Subsystems. Menlo Park CA. The Benjamin/Cummings Publishing Co. Inc.

-----. 1991. Object Oriented Design With. Applications. Redwood City CA. The Benjamin/Cummings Publishing Co. Inc.

Bralick, Capt William A. Jr. 1988. An Examination of the Theoretical Foundations of the Object-Oriented Paradigm. MS thesis, AFIT/GCS/MA/88M-01. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1988.

Coad, P. and E. Yourdon. 1990. Object-Oriented Analysis. Englewood Cliffs NJ. Prentice Hall.

Davis, Alan M. 1990. Software Requirements Analysis & Specification. Englewood Cliffs NJ. Prentice-Hall, Inc.

Goldberg, Jay H. 1990. "The Pentagon's Software Crisis Jeopardizes Key Weapon Programs," Armed Forces Journal International. June:60-70.

Henderson-Sellers, Brian and Julian M. Edwards. 1990. "The Object-Oriented Systems Life Cycle," Communications of the ACM, 33, 9:143-159.

Ince, D. C. 1988. An Introduction to Discrete Mathematics and Formal System Specification. Clarendon Press NY.

MacLennan, Bruce J. 1982. "Values and Objects in Programming Languages," SIGPLAN Notices, 17, 12:70-79.

Reasoning Systems, Inc. 1990. Refine Users Guide (version 3.0). 3260 Hillview Ave., Palo Alto CA.

Rombach, Dieter H. 1989. "Software Specifications: A Framework," SEI Curriculum Module SEI-CM-11-2.0. Carnegie Mellon University.

Shlaer, Sally and Stephen J. Mellor. 1988. Object-Oriented Systems Analysis: Modeling the World in Data. Englewood Cliffs NJ. Prentice Hall.

-----. 1989. "An Object-Oriented Approach to Domain Analysis," Software Engineering Notes, 14, 5:66-77.

Sommerville, Ian. 1989. Software Engineering (third edition). Wokingham, England. Addison-Wesley Publishing Co..

Spivey, J. M. 1989. The Z Notation A Reference Manual. United Kingdom. Prentice Hall International.

Umphress, David A. 1990. Class handout distributed in CSCE 593, Systems and Software Analysis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH.

-----. 1991. Associate Professor of Computer Science. Personal Correspondence. Department of Electrical and Computer Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH.

Umphress, David A. and Steven G. March. 1991 (In press). "Object-Oriented Requirements Determination," Journal of Object-Oriented Programming.

Yelland, P. M. 1989. "First Steps Towards Fully Abstract Semantics for Object-Oriented Languages," The Computer Journal, 32, 4:290-296.

Vita

Captain Andrew D. Boyd was born on 14 March 1960 in Kansas City, Missouri. He graduated from Mannheim American High School, Germany in 1978 and attended the University of Kansas, Lawrence, graduating with a Bachelor of Science in Chemical Engineering in December 1983. He attended the Air Force Officers Training School (OTS), class 85-02, and received his commission on 7 November 1984. His first tour of duty was at the Los Angeles Air Force Base, California. He began as a business/financial manager for the Space Test Program and later as a payload program manager for the Space Shuttle Program Office. He managed the STP-1 payload which flew on STS-39 in May 1991. Captain Boyd's next tour was at Wright-Patterson Air Force Base's Electronic Combat and Reconnaissance Program Office where he was the program manager of the Air Force Electronic Warfare Evaluation Simulator (AFEWES) upgrades program until entering the School of Systems and Logistics, Air Force Institute of Technology, in May 1990, in the first Graduate Software Systems Management (GSS) program.

Permanent Address: c/o Robert Boyd
1309 55th St.
Blue Springs, Mo.
64015

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 91	Master's Thesis

4. TITLE AND SUBTITLE

A FORMAL DEFINITION OF THE OBJECT-ORIENTED PARADIGM FOR REQUIREMENTS ANALYSIS

5. FUNDING NUMBERS

6. AUTHOR(S)

Andrew D. Boyd, Captain USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology,
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GSS/ENG/91D-3

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release: distribution unlimited

12b. DISTRIBUT ON CODE

13. ABSTRACT (*Maximum 200 words*)

This paper develops a formal definition of the Object-Oriented paradigm for requirements analysis. The literature was surveyed for both formal and informal methods for conducting an Object-Oriented Requirements Analysis (OORA). The informal methods reviewed are: Bailin's, Shlaer and Mellor's, Booch's, and Coad and Yourdon's. The formal methods reviewed are: Bralick's, Z, and REFINE. None of the methods were found to be adequate for doing an OORA. A formal definition of an OORA, based on the concept of classes, is developed. The definition itself is presented as set and relation theory. A supporting graphical representation is also developed and presented. The graphical method allows a system to be successfully leveled. The formalism is validated by applying it to the Air Traffic Control (ATC) simulation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Object-Oriented Paradigm, Requirements Analysis, Formal Definition	90
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

AFIT Control Number AFIT/GSS/ENG/91D-3

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to determine the potential for current and future applications of AFIT thesis research. Please return completed questionnaires to: AFIT/LSC, Wright-Patterson AFB OH 45433-6583.

1. Did this research contribute to a current research project?

   a. Yes b. No

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not researched it?

   a. Yes b. No

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency received by virtue of AFIT performing the research. Please estimate what this research would have cost in terms of manpower and/or dollars if it had been accomplished under contract or if it had been done in-house.

   Man Years _____ $_____

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3 above), what is your estimate of its significance?

   a. Highly Significant b. Significant c. Slightly Significant d. Of No Significance

5. Comments

_____ _____

Name and Grade Organization

_____ _____

Position or Title Address